SimpliChef, Token, ZapBSC & Broker

Smart Contract Audit Report
Prepared for SIMPLI FINANCE LAB

Date Issued:	Feb 1, 2022
Project ID:	AUDIT2022003
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022003
Version	v1.0
Client	SIMPLI FINANCE LAB
Project	SimpliChef, Token, ZapBSC & Broker
Auditor(s)	Patipon Suwanbol Puttimet Thammasaeng Ronnachai Chaipha
Author(s)	Ronnachai Chaipha
Reviewer	Suvicha Buakhom
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Feb 1, 2022	Full report	Ronnachai Chaipha

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by SIMPLI FINANCE LAB, Inspex team conducted an audit to verify the security posture of the SimpliChef, Token, ZapBSC & Broker smart contracts between Jan 13, 2022 and Jan 14, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of SimpliChef, Token, ZapBSC & Broker smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 high, 2 medium, 6 low, 2 very low, and 3 info-severity issues. With the project team's prompt response, 1 high issue was mitigated in the reassessment, while 1 low, 1 very low, and 1 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that SimpliChef, Token, ZapBSC & Broker smart contracts have sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

SIMPLI is an investment tool designed to optimize and allocate your assets in DeFi across all DApps and chains. SIMPLI uses Artificial Intelligence (AI) algorithms to analyze data and allocate your assets according to your preference and risk tolerance which can be done with just one simple click.

SimpliChef allows the platform's users to do yield farming, this means the users can stake tokens and earn reward ($SIMPLI). The platform also assists the users to conveniently convert any token to needed liquidity tokens in order to stake through ZapBSC. Furthermore, if the users want to gain more, the platform offers the compounded pools through the Broker.

Scope Information:

Project Name	SimpliChef, Token, ZapBSC & Broker
Website	https://app.simplifinance.io/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jan 13, 2022 - Jan 14, 2022
Reassessment Date	Jan 28, 2022

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: 4627062c76e7e065e63e5b7b07a1948525e71c0e)

Contract	Location (URL)
SimpliChef	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4627062c76/contracts/SimpliChef.sol
SimpliToken	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4627062c76/contracts/SimpliToken.sol
StratX2_PCS	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4627062c76/contracts/strategies/StratX2_PCS.sol
Broker	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4627062c76/contracts/broker/Broker.sol
ZapBSC	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4627062c76/contracts/zap/ZapBSC.sol

Reassessment: (Commit: 4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419)

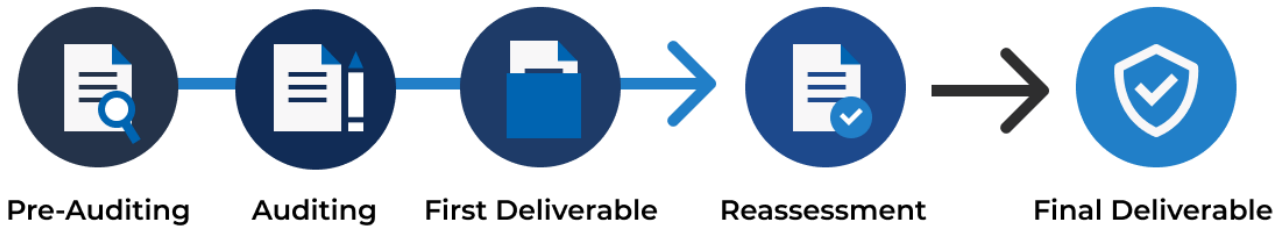
Contract	Location (URL)
SimpliChef	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4823d1ebbd/contracts/SimpliChef.sol
SimpliToken	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4823d1ebbd/contracts/SimpliToken.sol
StratX2_PCS	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4823d1ebbd/contracts/strategies/StratX2_PCS.sol
Broker	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4823d1ebbd/contracts/broker/Broker.sol
ZapBSC	https://github.com/SimpliFinanceLab/simpli-smart-contract/blob/4823d1ebbd/contracts/zap/ZapBSC.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Insufficient Logging for Privileged Functions
Invoking of Unreliable Smart Contract
Use of Upgradable Contract Design
Centralized Control of State Variable
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication

Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation
Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

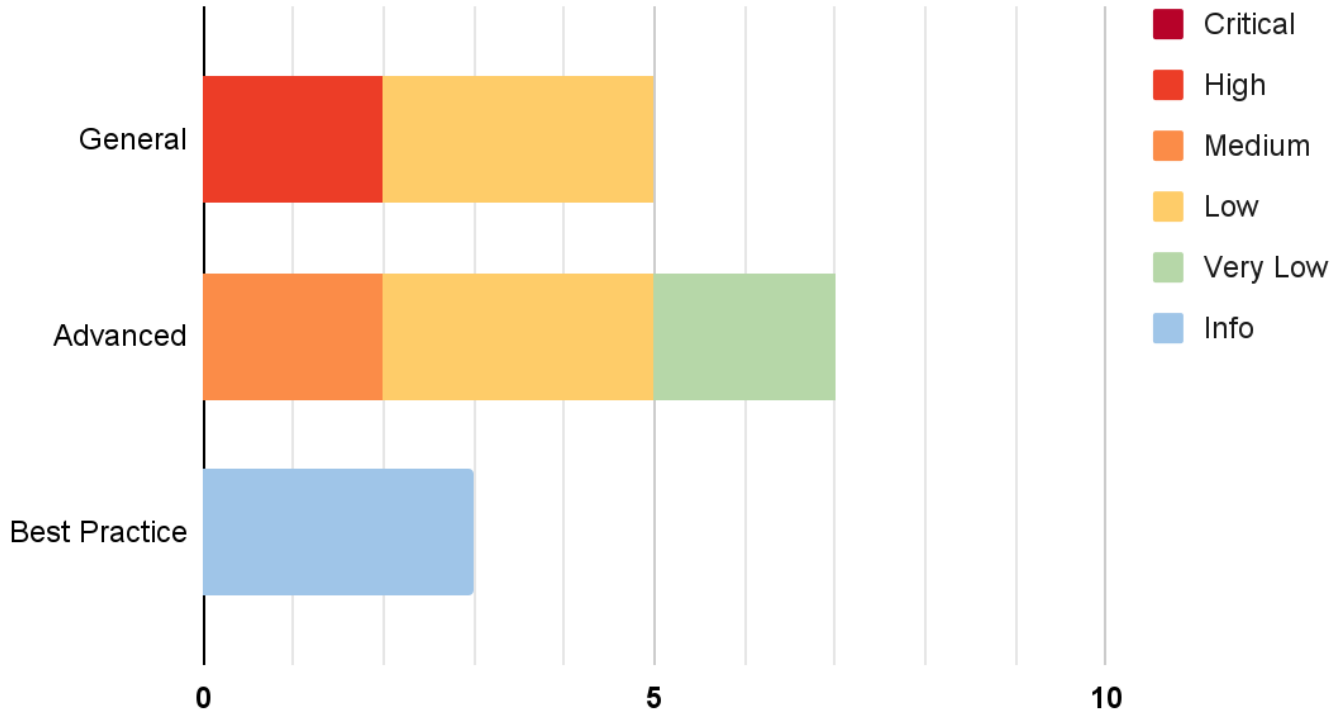
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

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

4. Summary of Findings

From the assessments, Inspex has found 15 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Centralized Control of State Variable	General	High	Resolved *
IDX-002	Use of Upgradable Contract Design	General	High	Resolved
IDX-003	Improper Reward Calculation on _withUpdate Parameter	Advanced	Medium	Resolved
IDX-004	Unchecked Token Transfer In on batchZapInBNB() Function	Advanced	Medium	Resolved
IDX-005	Transaction Ordering Dependence	General	Low	Acknowledged
IDX-006	Improper Fee Calculation on deposit() Function	Advanced	Low	Resolved
IDX-007	Design Flaw in massUpdatePools() Function	General	Low	Resolved
IDX-008	Improper Reward Calculation (Duplicated Strategy)	Advanced	Low	Resolved
IDX-009	Improper Compliance to the Tokenomics	Advanced	Low	Resolved
IDX-010	Insufficient Logging for Privileged Functions	General	Low	Resolved
IDX-011	Improper Allowance Checking in Approval Functions	Advanced	Very Low	Resolved
IDX-012	Missing Boundary State Variable Setting in Constructor	Advanced	Very Low	Acknowledged
IDX-013	Unnecessary Function Declaration	Best Practice	Info	Resolved
IDX-014	Inexplicit Solidity Compiler Version	Best Practice	Info	No Security Impact
IDX-015	Improper Function Visibility	Best Practice	Info	Resolved

* The mitigations or clarifications by SIMPLI FINANCE LAB can be found in Chapter 5.

5. Detailed Findings Information

5.1. Centralized Control of State Variable

ID	IDX-001
Target	SimpliChef SimpliToken StratX2_PCS Zap
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The controlling authorities can change the critical state variables to gain additional profit. For example, the contract owner can call the `setBroker()` function to set the broker address to be any address, allowing the contract owner to call the `withdrawOnlyBroker()` function to transfer the user's fund to the set address. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner only.
Status	Resolved * SIMPLI FINANCE LAB team has confirmed to mitigate this issue by applying a timelock mechanism with a minimum of 24 hours delay to the `SimpliChef` and the `ZapBSC` contracts. For the `StratX2` contract, the contract owner will be the `SimpliChef` contract, and the `onlyAllowGov` role will be applied by the timelock mechanism with a minimum of 24 hours delay. This can ensure that all the privilege functions will be delayed, giving a room for the platform users to act consequently.

5.1.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Contract	Function	Modifier
SimpliChef.sol (L:1492)	SimpliChef	setBroker()	onlyOwner
SimpliChef.sol (L:1508)	SimpliChef	add()	onlyOwner
SimpliChef.sol (L:1532)	SimpliChef	set()	onlyOwner
SimpliChef.sol (L:1751)	SimpliChef	withdrawOnlyBroker()	onlyOwner
SimpliChef.sol (L:1831)	SimpliChef	inCaseTokensGetStuck()	onlyOwner
StratX2_PCS.sol (L:2073)	StratX2_PCS	setSettings()	onlyAllowGov
StratX2_PCS.sol (L:2121)	StratX2_PCS	setGov()	onlyAllowGov
StratX2_PCS.sol (L:2126)	StratX2_PCS	setOnlyGov()	onlyAllowGov
StratX2_PCS.sol (L:2131)	StratX2_PCS	setUniRouterAddress()	onlyAllowGov
StratX2_PCS.sol (L:2140)	StratX2_PCS	setBuyBackAddress()	onlyAllowGov
StratX2_PCS.sol (L:2149)	StratX2_PCS	setRewardsAddress()	onlyAllowGov
StratX2_PCS.sol (L:2158)	StratX2_PCS	inCaseTokensGetStuck()	onlyAllowGov
StratX2_PCS.sol (L:2206)	StratX2_PCS	addMember()	onlyAllowGov
StratX2_PCS.sol (L:2221)	StratX2_PCS	removeMember()	onlyAllowGov
ZapBSC.sol (L:353)	ZapBSC	setRoutePairAddress()	onlyOwner
ZapBSC.sol (L:357)	ZapBSC	setNotFlip()	onlyOwner
ZapBSC.sol (L:365)	ZapBSC	removeToken()	onlyOwner
ZapBSC.sol (L:372)	ZapBSC	sweep()	onlyOwner
ZapBSC.sol (L:383)	ZapBSC	withdraw()	onlyOwner
ZapBSC.sol (L:392)	ZapBSC	setSafeSwapBNB()	onlyOwner

5.1.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the changes for a reasonable amount of time

5.2. Use of Upgradable Contract Design

ID	IDX-002
Target	ZapBSC
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity: High** **Impact: High** The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the users' funds anytime they want. **Likelihood: Medium** This action can be performed by the proxy owner without any restriction.
Status	**Resolved** SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by changing the inherited `OwnableUpgradeable` contract to the `Ownable` contract instead.

5.2.1. Description

Smart contracts are designed to be used as agreements that cannot be changed forever. When a smart contract is upgraded, the agreement can be changed from what was previously agreed upon.

ZapBSC.sol

```
13 contract ZapBSC is IZap, OwnableUpgradeable {
14     using SafeMath for uint256;
15     using SafeBEP20 for IBEP20;
```

As these smart contracts can be deployed through a proxy contract, they can be upgradable. Therefore, the logic of them can be modified by the owner anytime, making the smart contracts untrustworthy.

5.2.2. Remediation

Inspex suggests deploying the contracts without the proxy pattern or any solution that can make smart contracts upgradable.

However, if the upgradability is needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient length of time to delay the changes e.g., 1 days. This allows the platform users to monitor the timelock and is notified of the potential changes being done on the smart contracts

5.3. Improper Reward Calculation on _withUpdate Parameter

ID	IDX-003
Target	SimpliChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium When the `add()` and `set()` functions are called without updating the pools, the reward will be miscalculated than what it should be, leading to unfair reward distribution. **Likelihood:** Medium The issue occurs when the contract owner manipulates the pools through the `add()` and `set()` functions and the `_withUpdate` parameter is set as false.
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by removing the `_withUpdate` parameter and always calling the `massUpdatePools()` function.

5.3.1. Description

In the `SimpliChef` contract, it allows the users to do yield farming by depositing the specific token to the adjusted pool. The contract owner (`onlyOwner`) can add a new pool and modify the existing pool reward through the `add()` and the `set()` function respectively.

SimpliChef.sol

```
1505 // Add a new lp to the pool. Can only be called by the owner.
1506 // XXX DO NOT add the same LP token more than once. Rewards will be messed up
1507 if you do. (Only if want tokens are stored here.)
1508
1509 function add(
1510     uint256 _allocPoint,
1511     IERC20 _want,
1512     bool _withUpdate,
1513     address _strat
1514 ) public onlyOwner {
1515     if (_withUpdate) {
1516         massUpdatePools();
1517     }
1518     uint256 lastRewardBlock =
         block.number > startBlock ? block.number : startBlock;
1519     totalAllocPoint = totalAllocPoint.add(_allocPoint);
```

```
1520        poolInfo.push(
1521            PoolInfo({
1522                want: _want,
1523                allocPoint: _allocPoint,
1524                lastRewardBlock: lastRewardBlock,
1525                accSIMPLIPerShare: 0,
1526                strat: _strat
1527            })
1528        );
1529    }
```

SimpliChef.sol

```
1531    // Update the given pool's SIMPLI allocation point. Can only be called by the
        owner.
1532    function set(
1533        uint256 _pid,
1534        uint256 _allocPoint,
1535        bool _withUpdate
1536    ) public onlyOwner {
1537        if (_withUpdate) {
1538            massUpdatePools();
1539        }
1540        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
1541            _allocPoint
1542        );
1543        poolInfo[_pid].allocPoint = _allocPoint;
1544    }
```

The `add()` and the `set()` functions accept `_withUpdate` parameter to determine whether to update the current reward calculation of the pool contract immediately or not through the `massUpdatePools()` function.

The `massUpdatePools()` function will call `updatePool()` to update each pool's `accSIMPLIPerShare` state that is used for calculating the user's reward distribution.

SimpliChef.sol

```
1608    // Update reward variables of the given pool to be up-to-date.
1609    function updatePool(uint256 _pid) public {
1610        PoolInfo storage pool = poolInfo[_pid];
1611        if (block.number <= pool.lastRewardBlock) {
1612            return;
1613        }
1614        uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
1615        if (sharesTotal == 0) {
1616            pool.lastRewardBlock = block.number;
1617            return;
```

```
1618        }
1619        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1620        if (multiplier <= 0) {
1621            return;
1622        }
1623        uint256 SIMPLIReward =
1624            multiplier.mul(SIMPLIPerBlock).mul(pool.allocPoint).div(
1625                totalAllocPoint
1626            );
1627
1628        SIMPLIToken(SIMPLI).mint(
1629            owner(),
1630            SIMPLIReward.mul(ownerSIMPLIReward).div(1000)
1631        );
1632        SIMPLIToken(SIMPLI).mint(address(this), SIMPLIReward);
1633
1634        pool.accSIMPLIPerShare = pool.accSIMPLIPerShare.add(
1635            SIMPLIReward.mul(1e12).div(sharesTotal)
1636        );
1637        pool.lastRewardBlock = block.number;
1638    }
```

As a result, if the contract owner passes `_withUpdate` as `false`, the reward will be calculated incorrectly since the `totalAllocPoint` is updated, but the pending reward of all other pools is not updated immediately with new `totalAllocaPoint` value.

For example:

Assuming that at block 1010000, `SIMPLIPerBlock` is set to 10 $SIMPLI per block, pool 0 `allocPoint` is set to 300, `totalAllocPoint` is set to 9605, and `pool.lastRewardBlock` is set to 1010000.

Block	Action
1010000	All pools' rewards are updated
1020000	A new pool is added using the add() function, causing the totalAllocPoint to be changed from 9605 to 10000
1030000	The pools' rewards are updated once again

From current logic, the total rewards allocated to the pool 0 during block 1010000 to block 1030000 is equal to 6,000.00 $SIMPLI calculated using the following equation:

```
multiplier = 1030000 - 1010000
SIMPLIReward = multiplier * SIMPLIPerBlock * pool 0 allocPoint /
totalAllocPoint
             = 20000 * 10 * 300 / 10,000 = 6,000.00
```

However, the rewards should be calculated by accounting for the original `totalAllocPoint` value during the period when it is not yet updated as follow:

- from block 1010000 to block 1020000, with a proportion of 300/9,605 = 3,123.37 $SIMPLI
- from block 1020000 to block 1030000, with a proportion of 300/10,000 = 3,000.00 $SIMPLI

The correct total $SIMPLI rewards is 6,123.37 $SIMPLI, which is different from the miscalculated reward by 123.37 $SIMPLI

5.3.2. Remediation

Inspex suggests removing the `_withUpdate` parameter and always calling the `massUpdatePools()` before updating `totalAllocPoint` variable as shown in the following examples:

SimpliChef.sol

```
1505 // Add a new lp to the pool. Can only be called by the owner.
1506 // XXX DO NOT add the same LP token more than once. Rewards will be messed up
     if you do. (Only if want tokens are stored here.)
1507
1508 function add(
1509     uint256 _allocPoint,
1510     IERC20 _want,
1511     address _strat
1512 ) public onlyOwner {
1513     massUpdatePools();
1514     uint256 lastRewardBlock =
1515         block.number > startBlock ? block.number : startBlock;
1516     totalAllocPoint = totalAllocPoint.add(_allocPoint);
1517     poolInfo.push(
1518         PoolInfo({
1519             want: _want,
1520             allocPoint: _allocPoint,
1521             lastRewardBlock: lastRewardBlock,
1522             accSIMPLIPerShare: 0,
1523             strat: _strat
1524         })
1525     );
1526 }
```

SimpliChef.sol

```
1531 // Update the given pool's SIMPLI allocation point. Can only be called by the
     owner.
1532 function set(
1533     uint256 _pid,
1534     uint256 _allocPoint,
1535 ) public onlyOwner {
1536     massUpdatePools();
1537     totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
1538         _allocPoint
1539     );
1540     poolInfo[_pid].allocPoint = _allocPoint;
1541 }
```

5.4. Unchecked Token Transfer In on batchZapInBNB() Function

ID	IDX-004
Target	ZapBSC
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Medium **Impact:** Medium The improper amount validation of `batchZapInBNB()` function lead to anyone can spend the leftover $BNB freely, for example, transfer the leftover $BNB to the specific wallet address. **Likelihood:** Medium By design, there is no $BNB stored in the ZapBSC contract. However, there will be accumulated $BNB left from other operations, leading to a scenario that the attacker can transfer those leftover $BNB freely.
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419`. Hence, the sum of `_amounts` must equal to $BNB that the `msg.sender` sends to (tx.value).

5.4.1. Description

The `batchZapInBNB()` function is used for bulk executing the `_swapBNBToFlip()` function in order to swap $BNB to multiple flip tokens (LP token) in one transaction.

ZapBSC.sol

```
130 function batchZapInBNB(uint256[] memory _amounts, address[] memory _to)
    external payable {
131     require(_amounts.length == _to.length, "Amounts and addresses don't
    match");
132
133     for (uint256 i=0; i < _amounts.length; i++) {
134         _swapBNBToFlip(_to[i], _amounts[i], msg.sender);
135     }
136 }
```

However, there is no input validation whether the amount of transfer $BNB is equal to the sum amount of `_amounts` to flip or not. Therefore, anyone can call this function to spend the leftover of $BNB in this contract freely.

5.4.2. Remediation

Inspex suggests adding an input validation to ensure the total of the `_amounts` array is equal to the `msg.value` (transferred $BNB amount).

ZapBSC.sol

```
130 function batchZapInBNB(uint256[] memory _amounts, address[] memory _to)
    external payable {
131     require(_amounts.length == _to.length, "Amounts and addresses don't
    match");
132     uint256 sumAmount = 0;
133
134     for (uint256 i=0; i < _amounts.length; i++) {
135         sumAmount = sumAmount + _amounts[i];
136         _swapBNBToFlip(_to[i], _amounts[i], msg.sender);
137     }
138     require(sumAmount == msg.value, "Amount unmatched");
139 }
```

5.5. Transaction Ordering Dependence

ID	IDX-005
Target	StratX2_PCS
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** Low **Impact:** Low The front-running attack can be performed, resulting in a bad swapping rate. However, due to the fact that the compounding will be frequently executed, this means the amount to compound will be low, making the gain for the attack to be insignificant. **Likelihood:** Medium Anyone who understands the EVM architecture can simply monitor the BSC's transaction pool and perform this attack.
Status	**Acknowledged** SIMPLI FINANCE LAB team has acknowledged this issue. However, the amount of compound is quite low due to the business design, which means the risk is acceptable.

5.5.1. Description

In the `StratX2_PCS` contract, there is an internal function named `_safeSwap()` that will be called by the `earn()`, the `buyBack()`, and the `convertDustToEarned()` functions to swap earned reward tokens into the wanted tokens (LP token) for compounding the rewards.

StratX2_PCS.sol

```
2180 function _safeSwap(
2181     address _uniRouterAddress,
2182     uint256 _amountIn,
2183     uint256 _slippageFactor,
2184     address[] memory _path,
2185     address _to,
2186     uint256 _deadline
2187 ) internal virtual {
2188     uint256[] memory amounts =
2189         IPancakeRouter02(_uniRouterAddress).getAmountsOut(_amountIn, _path);
2190     uint256 amountOut = amounts[amounts.length.sub(1)];
2191
2192     IPancakeRouter02(_uniRouterAddress)
2193         .swapExactTokensForTokensSupportingFeeOnTransferTokens(
2194         _amountIn,
```

```
2195            amountOut.mul(_slippageFactor).div(1000),
2196            _path,
2197            _to,
2198            _deadline
2199        );
2200    }
```

From the source code shown above, the value of `amountOut` comes from the external contract by calling the `getAmountsOut()` function. So, the slippage will be calculated from the latest price, which means the `_slippageFactor` parameter cannot prevent the price impact from being exceeded as it should be from design.

5.5.2. Remediation

Inspex suggests calculating the expected `amountOut` with the token price fetched from the price oracle. For example, this can be done by implementing and deploying a TWAP oracle[2].

5.6. Improper Fee Calculation on deposit() Function

ID	IDX-006
Target	StratX2_PCS
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low The first user, who deposits the token through the `deposit()` function, will be excluded from the fee collection. This means the platform will not be able to claim the fee in this part. **Likelihood:** Medium This issue happens on the first `deposit()` function execution, which can be called by anyone.
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by adding the fee calculation for the first deposit.

5.6.1. Description

In the `deposit()` function on the abstract `StratX2` contract inherited by the `StratX2_PCS` contract, the `sharesAdded` variable is calculated from `_wantAmt` parameter. In every deposit, The `sharesAdded` must subtract for the entrance fee that is calculated from the `entranceFeeFactor` and `entranceFeeFactorMax` variable that is shown in the following source code:

StratX2_PCS.sol

```
1785 function deposit(address _userAddress, uint256 _wantAmt)
1786     public
1787     virtual
1788     onlyOwner
1789     nonReentrant
1790     whenNotPaused
1791     returns (uint256)
1792 {
1793     require(!isMember(_userAddress), "!auth");
1794
1795     IERC20(wantAddress).safeTransferFrom(
1796         address(msg.sender),
1797         address(this),
1798         _wantAmt
```

```
1799        );
1800
1801        uint256 sharesAdded = _wantAmt;
1802        if (wantLockedTotal > 0 && sharesTotal > 0) {
1803            sharesAdded = _wantAmt
1804                .mul(sharesTotal)
1805                .mul(entranceFeeFactor)
1806                .div(wantLockedTotal)
1807                .div(entranceFeeFactorMax);
1808        }
1809        sharesTotal = sharesTotal.add(sharesAdded);
1810
1811        if (isAutoComp) {
1812            _farm();
1813        } else {
1814            wantLockedTotal = wantLockedTotal.add(_wantAmt);
1815        }
1816
1817        return sharesAdded;
1818    }
```

The deposit amount will be deducted by the fee only when the `wantLockedTotal` and `sharesTotal` are more than 0. Since the value of `wantLockedTotal` and `sharesTotal` will be set to 0 by default, the first wallet who executes the `deposit()` function will not pay any fee.

StratX2.sol

```
1735    uint256 public wantLockedTotal = 0;
1736    uint256 public sharesTotal = 0;
```

5.6.2. Remediation

Inspex suggests adding the fee calculation for the first deposit by adding the following source code shown in lines 1808 - 1813:

StratX2_PCS.sol

```
1785    function deposit(address _userAddress, uint256 _wantAmt)
1786        public
1787        virtual
1788        onlyOwner
1789        nonReentrant
1790        whenNotPaused
1791        returns (uint256)
1792    {
1793        require(!isMember(_userAddress), "!auth");
1794
1795        IERC20(wantAddress).safeTransferFrom(
```

```
1796            address(msg.sender),
1797            address(this),
1798            _wantAmt
1799        );
1800
1801        uint256 sharesAdded = _wantAmt;
1802        if (wantLockedTotal > 0 && sharesTotal > 0) {
1803            sharesAdded = _wantAmt
1804                .mul(sharesTotal)
1805                .mul(entranceFeeFactor)
1806                .div(wantLockedTotal)
1807                .div(entranceFeeFactorMax);
1808          } else if (wantLockedTotal == 0 && sharesTotal == 0){
1809                // init case: calculate the entraceFeeFactor
1810                sharesAdded = _wantAmt
1811                        .mul(entranceFeeFactor)
1812                        .div(entranceFeeFactorMax);
1813          }
1814        sharesTotal = sharesTotal.add(sharesAdded);
1815
1816        if (isAutoComp) {
1817            _farm();
1818        } else {
1819            wantLockedTotal = wantLockedTotal.add(_wantAmt);
1820        }
1821
1822        return sharesAdded;
1823    }
```

5.7. Design Flaw in massUpdatePools() Function

ID	IDX-007
Target	SimpliChef
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePools()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `poolInfo` size will be raised until the `massUpdatePools()` function is eventually unusable.
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by checking `allocPoint` before updating pool so that the pool with `allocPoint` as 0 will be skipped.

5.7.1. Description

The `massUpdatePools()` function executes the `updatePool()` function, which is a state modifying function for all added pools as shown below:

SimpliChef.sol

```
1600 // Update reward variables for all pools. Be careful of gas spending!
1601 function massUpdatePools() public {
1602     uint256 length = poolInfo.length;
1603     for (uint256 pid = 0; pid < length; ++pid) {
1604         updatePool(pid);
1605     }
1606 }
```

With the current design, the added pools cannot be removed. They can only be disabled by setting the `pool.allocPoint` to 0. Even if a pool is disabled, the `updatePool()` function for this pool is still called. Therefore, if new pools continue to be added to this contract, the `poolInfo.length` will continue to grow and this function will eventually be unusable due to excessive gas usage.

5.7.2. Remediation

Inspex suggests making the contract capable of removing unnecessary or ended pools to reduce the loop round in the `massUpdatePools()` function.

5.8. Improper Reward Calculation (Duplicated Strategy)

ID	IDX-008
Target	SimpliChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The reward of the pool that has the same strategy contract address token will be slightly lower than what it should be, causing the users to gain less reward, which may result in reputation damage to the platform. **Likelihood:** Low It is unlikely that there will be pools with the same strategy contract added by the contract owner. This is because on the adding of a new pool, a new contract should be deployed as the strategy according to the business design.
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by validating strategy whether it has been used by the other pools before or not.

5.8.1. Description

In the `SimpliChef` contract, a new pool can be added using the `add()` function. The strategy contract for the new pool is defined using the `_strat` variable; however, there is no additional checking whether the `_strat` is the same as the other pool or not.

SimpliChef.sol

```
1505 // Add a new lp to the pool. Can only be called by the owner.
1506 // XXX DO NOT add the same LP token more than once. Rewards will be messed up
     if you do. (Only if want tokens are stored here.)
1507
1508 function add(
1509     uint256 _allocPoint,
1510     IERC20 _want,
1511     bool _withUpdate,
1512     address _strat
1513 ) public onlyOwner {
1514     if (_withUpdate) {
1515         massUpdatePools();
1516     }
1517     uint256 lastRewardBlock =
```

```
1518            block.number > startBlock ? block.number : startBlock;
1519        totalAllocPoint = totalAllocPoint.add(_allocPoint);
1520        poolInfo.push(
1521            PoolInfo({
1522                want: _want,
1523                allocPoint: _allocPoint,
1524                lastRewardBlock: lastRewardBlock,
1525                accSIMPLIPerShare: 0,
1526                strat: _strat
1527            })
1528        );
1529    }
```

When the `_strat` is the same strategy as the other pools, reward calculation for that pool in the `updatePool()` function can be incorrect. This is because the current share of the `sharesTotal` in the strategy is used in the calculation of the reward. Since the `_strat` is duplicated, the value of `sharesTotal` will be mixed up with other pools, causing the reward of that pool to be less than what it should be.

SimpliChef.sol

```
1608    // Update reward variables of the given pool to be up-to-date.
1609    function updatePool(uint256 _pid) public {
1610        PoolInfo storage pool = poolInfo[_pid];
1611        if (block.number <= pool.lastRewardBlock) {
1612            return;
1613        }
1614        uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
1615        if (sharesTotal == 0) {
1616            pool.lastRewardBlock = block.number;
1617            return;
1618        }
1619        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1620        if (multiplier <= 0) {
1621            return;
1622        }
1623        uint256 SIMPLIReward =
1624            multiplier.mul(SIMPLIPerBlock).mul(pool.allocPoint).div(
1625                totalAllocPoint
1626            );
1627
1628        SIMPLIToken(SIMPLI).mint(
1629            owner(),
1630            SIMPLIReward.mul(ownerSIMPLIReward).div(1000)
1631        );
1632        SIMPLIToken(SIMPLI).mint(address(this), SIMPLIReward);
1633
1634        pool.accSIMPLIPerShare = pool.accSIMPLIPerShare.add(
```

```
1635            SIMPLIReward.mul(1e12).div(sharesTotal)
1636        );
1637        pool.lastRewardBlock = block.number;
1638    }
```

5.8.2. Remediation

Before adding a new pool, Inspex suggests applying a checking mechanism to validate whether the strategy is being used or not, for example, the mapping state variable can be added to the `SimpliChef` contract to keep track of the used strategy address.

SimpliChef.sol

```
1   mapping(address => bool) private usedStrategy;
```

SimpliChef.sol

```
1505 // Add a new lp to the pool. Can only be called by the owner.
1506 // XXX DO NOT add the same LP token more than once. Rewards will be messed up
     if you do. (Only if want tokens are stored here.)
1507
1508 function add(
1509     uint256 _allocPoint,
1510     IERC20 _want,
1511     bool _withUpdate,
1512     address _strat
1513 ) public onlyOwner {
1514     require(!usedStrategy[_strat], '_strat is already used');
1515     if (_withUpdate) {
1516         massUpdatePools();
1517     }
1518     uint256 lastRewardBlock =
1519         block.number > startBlock ? block.number : startBlock;
1520     totalAllocPoint = totalAllocPoint.add(_allocPoint);
1521     poolInfo.push(
1522         PoolInfo({
1523             want: _want,
1524             allocPoint: _allocPoint,
1525             lastRewardBlock: lastRewardBlock,
1526             accSIMPLIPerShare: 0,
1527             strat: _strat
1528         })
1529     );
1530     usedStrategy[_strat] = true;
1531 }
```

Please note that the remediation for other issues has not been applied to the example above yet.

5.9. Improper Compliance to the Tokenomics

ID	IDX-009
Target	SimpliChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low $SIMPLI can be minted and slightly exceed the expected amount compared to the platform's business design. This can cause reputation damage to the platform. **Likelihood:** Medium It is likely that the $SIMPLI will exceed the total supply limit since there is no mechanism to prevent it. However, after the amount is exceeded, $SIMPLI will be unable to be minted anymore (exceeded total supply).
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by checking the `SIMPLIReward` that it cannot be minted greater than the token total supply.

5.9.1. Description

The `updatePool()` function is used for calculating the amount of minted reward in order to distribute for the users. The `multiplier` variable is one of the factors that applies to the reward distribution formula.

SimpliChef.sol

```
1608 // Update reward variables of the given pool to be up-to-date.
1609 function updatePool(uint256 _pid) public {
1610     PoolInfo storage pool = poolInfo[_pid];
1611     if (block.number <= pool.lastRewardBlock) {
1612         return;
1613     }
1614     uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
1615     if (sharesTotal == 0) {
1616         pool.lastRewardBlock = block.number;
1617         return;
1618     }
1619     uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1620     if (multiplier <= 0) {
1621         return;
1622     }
1623     uint256 SIMPLIReward =
```

```
1624            multiplier.mul(SIMPLIPerBlock).mul(pool.allocPoint).div(
1625                totalAllocPoint
1626            );
1627
1628        SIMPLIToken(SIMPLI).mint(
1629            owner(),
1630            SIMPLIReward.mul(ownerSIMPLIReward).div(1000)
1631        );
1632        SIMPLIToken(SIMPLI).mint(address(this), SIMPLIReward);
1633
1634        pool.accSIMPLIPerShare = pool.accSIMPLIPerShare.add(
1635            SIMPLIReward.mul(1e12).div(sharesTotal)
1636        );
1637        pool.lastRewardBlock = block.number;
1638    }
```

The `getMultiplier()` function takes the `pool.lastRewardBlock` and the `block.number` as the `_from` and the `_to` parameters respectively to determine how many blocks have passed.

SimpliChef.sol

```
1547 function getMultiplier(uint256 _from, uint256 _to)
1548     public
1549     view
1550     returns (uint256)
1551 {
1552     if (IERC20(SIMPLI).totalSupply() >= SIMPLIMaxSupply) {
1553         return 0;
1554     }
1555     return _to.sub(_from);
1556 }
```

However, the checking mechanism validates only whether the current total supply exceeds the `SIMPLIMaxSupply` or not (pre-check). This means the last reward distribution can make the `SimpliChef` contract mint more $SIMPLI than the allowed total supply.

5.9.2. Remediation

Inspex suggests adding a mechanism to validate that after the minting process is done, the amount of minted $SIMPLI must not exceed the total supply, for example:

SimpliChef.sol

```
1608 // Update reward variables of the given pool to be up-to-date.
1609 function updatePool(uint256 _pid) public {
1610     PoolInfo storage pool = poolInfo[_pid];
1611     if (block.number <= pool.lastRewardBlock) {
1612         return;
```

```
1613        }
1614        uint256 sharesTotal = IStrategy(pool.strat).sharesTotal();
1615        if (sharesTotal == 0) {
1616            pool.lastRewardBlock = block.number;
1617            return;
1618        }
1619        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
1620        if (multiplier <= 0) {
1621            return;
1622        }
1623        uint256 SIMPLIReward =
1624            multiplier.mul(SIMPLIPerBlock).mul(pool.allocPoint).div(
1625                totalAllocPoint
1626            );
1627        SIMPLIReward = (SIMPLIReward + IERC20(SIMPLI).totalSupply()) >
    SIMPLIMaxSupply ? (SIMPLIMaxSupply - IERC20(SIMPLI).totalSupply()) :
    SIMPLIReward;
1628        SIMPLIToken(SIMPLI).mint(
1629            owner(),
1630            SIMPLIReward.mul(ownerSIMPLIReward).div(1000)
1631        );
1632        SIMPLIToken(SIMPLI).mint(address(this), SIMPLIReward);
1633
1634        pool.accSIMPLIPerShare = pool.accSIMPLIPerShare.add(
1635            SIMPLIReward.mul(1e12).div(sharesTotal)
1636        );
1637        pool.lastRewardBlock = block.number;
1638    }
```

5.10. Insufficient Logging for Privileged Functions

ID	IDX-010
Target	SimpliChef SimpliToken StratX2_PCS Zap
Category	General Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Medium It is likely that these functions will be executed by authorized parties as they are used to accomplish the business flow.
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by adding the log emitting for all privileged functions.

5.10.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts on the platform.

For example, the owner can set the broker address by executing the `setBroker()` function in the `SimpliChef` contract, the new set broker address can execute all privileged functions for the broker such as the `emergencyWithdraw()` function and no events are emitted.

The privileged functions without sufficient logging are as follows:

Target	Contract	Function	Modifier
SimpliChef.sol (L:1492)	SimpliChef	setBroker()	onlyOwner
SimpliChef.sol (L:1508)	SimpliChef	add()	onlyOwner
SimpliChef.sol (L:1532)	SimpliChef	set()	onlyOwner
SimpliChef.sol (L:1831)	SimpliChef	inCaseTokensGetStuck()	onlyOwner

StratX2_PCS.sol (L:2158)	StratX2_PCS	inCaseTokensGetStuck()	onlyAllowGov
StratX2_PCS.sol (L:2168)	StratX2_PCS	wrapBNB()	onlyAllowGov
StratX2_PCS.sol (L:2206)	StratX2_PCS	addMember()	onlyAllowGov
StratX2_PCS.sol (L:2221)	StratX2_PCS	removeMember()	onlyAllowGov
ZapBSC.sol (L:353)	ZapBSC	setRoutePairAddress()	onlyOwner
ZapBSC.sol (L:357)	ZapBSC	setNotFlip()	onlyOwner
ZapBSC.sol (L:365)	ZapBSC	removeToken()	onlyOwner
ZapBSC.sol (L:372)	ZapBSC	sweep()	onlyOwner
ZapBSC.sol (L:383)	ZapBSC	withdraw()	onlyOwner
ZapBSC.sol (L:392)	ZapBSC	setSafeSwapBNB()	onlyOwner

5.10.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

SimpliChef.sol

```
1492 event SetBroker(address _broker);
1493 function setBroker(address _broker) public onlyOwner {
1494     broker = _broker;
1495     emit SetBroker(_broker);
1496 }
```

5.11. Improper Allowance Checking in Approval Functions

ID	IDX-011
Target	Broker ZapBSC
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low The token transfer flow from any operation could be stuck if the given allowance value is deducted until it is less than the input amount, resulting in denial of service of that contract. However, with the current design, it can be fixed by using the exact amount of token input in order to reset the allowance from zero to max `uint256` value again. **Likelihood:** Low It is very unlikely that the given allowance will be reduced from a max value of `uint256` until it is less than the swap amount.
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by checking the usage amount before increasing allowance.

5.11.1. Description

In the `Broker` and the `ZapBSC` contracts, before any related token transfer action occurs, the contract will execute the function that checks the current token allowance of the target address and call the `safeApprove()` function to approve the target address to manage the token freely, for example:

A user calls the `Broker` contract in order to deposit tokens to the `SimpliChef` contract through the `zapInTokenAndDeposit()` function.

Broker.sol

```
53  function zapInTokenAndDeposit(
54      address _from,
55      uint256[] memory _amounts,
56      address[] memory _to,
57      uint256[] memory _pid,
58      address _beneficiary
59  ) external {
60      require(_amounts.length == _to.length, "Amount and address lengths don't
61  match");
62      require(_to.length == _pid.length, "Address and pid lengths don't match");
```

```
63      _approveTokenForZap(_from);
64      uint256 sumAmount = 0;
65      for (uint256 i=0; i < _amounts.length; i++){
66          sumAmount = sumAmount.add(_amounts[i]);
67      }
68      IBEP20(_from).safeTransferFrom(msg.sender, address(this), sumAmount);
69      for (uint256 i=0; i < _amounts.length; i++) {
70          (, , uint256 LPAmount) = zap.zapInToken(_from, _amounts[i], _to[i]);
71          _approveTokenForSimpliChef(_to[i]);
72          simplichef.depositOnlyBroker(_pid[i], LPAmount, _beneficiary);
73      }
    }
```

The `Broken` contract then executes the `_approveTokenForSimpliChef()` to verify the current allowance and set the new allowance to the `simpliechef` (`SimpliChef` contract).

Broker.sol

```
39  function _approveTokenForSimpliChef(address token) private {
40      if (IBEP20(token).allowance(address(this), address(simplichef)) == 0) {
41          IBEP20(token).safeApprove(address(simplichef), type(uint256).max);
42      }
43  }
```

Following the source code above, the contract will only execute `safeApprove()` if the allowance value is 0, which means it will be executed only for the first time. Since, every transfer action will deduct the `Broker`'s allowance of the `simplichef`, the allowance will eventually become less than the current given allowance, resulting in denial of service of the `Broker` contract.

The following table shows the functions that involve the approval operation:

Contract	Function	Involved Function
Broker	_approveTokenForZap()	zapInTokenAndDeposit(), withdrawAndZapOut()
Broker	_approveTokenForSimpliChef()	zapInTokenAndDeposit(), zapInBNBAndDeposit()
Broker	_approveTokenForBroker()	-
ZapBSC	_approveTokenIfNeeded()	zapInToken(), batchZapInToken(), zapOut(), zapOutToToken(), _swapBNBToFlip()

5.11.2. Remediation

Inspex suggests increasing the sufficient allowance every time with users input amount instead of approving token allowance to the maximum value of `uint256`, for example:

Broker.sol

```
52  // zapAndDeposit
53  function zapInTokenAndDeposit(
54      address _from,
55      uint256[] memory _amounts,
56      address[] memory _to,
57      uint256[] memory _pid,
58      address _beneficiary
59  ) external {
60      require(_amounts.length == _to.length, "Amount and address lengths don't
    match");
61      require(_to.length == _pid.length, "Address and pid lengths don't match");
62      uint256 sumAmount = 0;
63      for (uint256 i=0; i < _amounts.length; i++){
64          sumAmount = sumAmount.add(_amounts[i]);
65      }
66      IBEP20(_from).safeIncreaseAllowance(address(zap), sumAmount);
67      IBEP20(_from).safeTransferFrom(msg.sender, address(this), sumAmount);
68      for (uint256 i=0; i < _amounts.length; i++) {
69          (, , uint256 LPAmount) = zap.zapInToken(_from, _amounts[i], _to[i]);
70          IBEP20(_to[i]).safeIncreaseAllowance(address(simplichef), _amounts[i]);
71          simplichef.depositOnlyBroker(_pid[i], LPAmount, _beneficiary);
72      }
73  }
```

5.12. Missing Boundary State Variable Setting in Constructor

ID	IDX-012
Target	StratX2_PCS
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Very Low **Impact:** Low Unexpected fee factor value can lead to incorrect calculation, resulting in unforeseen outcomes, for example, transactions being reverted. **Likelihood:** Low It is unlikely that the fee factors will be set with unexpected value. Even though it is set with unexpected value, the allowed party can execute the `setSettings()` function to set the fee factors with the proper value from the platform's business design.
Status	**Acknowledged** SIMPLI FINANCE LAB team has acknowledged this issue. However, the risk is quite low, because it is unlikely that the fee factors will be set with unexpected value and it can be set again to overwrite that unexpected value setting.

5.12.1. Description

In the `StratX2_PCS` contract, there are few factors that are used to describe the amount of fee to be deducted during the calculation according to the platform's business flow, for example, the `distributeFees()` function is used to deduct a portion of reward token from the `earn()` function.

StratX2_PCS.sol

```
1886 // 1. Harvest farm tokens
1887 // 2. Converts farm tokens into want tokens
1888 // 3. Deposits want tokens
1889
1890 function earn() public virtual nonReentrant whenNotPaused {
1891     require(isAutoComp, "!isAutoComp");
1892     if (onlyGov) {
1893         require(msg.sender == govAddress, "!gov");
1894     }
1895
1896     // Harvest farm tokens
1897     _unfarm(0);
1898
1899     if (earnedAddress == wbnbAddress) {
1900         _wrapBNB();
```

```
1901        }
1902
1903        // Converts farm tokens into want tokens
1904        uint256 earnedAmt = IERC20(earnedAddress).balanceOf(address(this));
1905
1906        earnedAmt = distributeFees(earnedAmt);
1907        earnedAmt = buyBack(earnedAmt);
```

The deducted amount relies on the `controllerFee` and `controllerFeeMax` state variable and that amount will be sent to the `rewardAddress`.

StratX2_PCS.sol

```
2002 function distributeFees(uint256 _earnedAmt)
2003     internal
2004     virtual
2005     returns (uint256)
2006 {
2007     if (_earnedAmt > 0) {
2008         // Performance fee
2009         if (controllerFee > 0) {
2010             uint256 fee =
2011                 _earnedAmt.mul(controllerFee).div(controllerFeeMax);
2012             IERC20(earnedAddress).safeTransfer(rewardsAddress, fee);
2013             _earnedAmt = _earnedAmt.sub(fee);
2014         }
2015     }
2016
2017     return _earnedAmt;
2018 }
```

All the fee factors can be set anytime through the `setSettings()` function.

StratX2_PCS.sol

```
2073 function setSettings(
2074     uint256 _entranceFeeFactor,
2075     uint256 _withdrawFeeFactor,
2076     uint256 _controllerFee,
2077     uint256 _buyBackRate,
2078     uint256 _slippageFactor
2079 ) public virtual onlyAllowGov {
2080     require(
2081         _entranceFeeFactor >= entranceFeeFactorLL,
2082         "_entranceFeeFactor too low"
2083     );
2084     require(
2085         _entranceFeeFactor <= entranceFeeFactorMax,
```

```
2086            "_entranceFeeFactor too high"
2087        );
2088        entranceFeeFactor = _entranceFeeFactor;
2089
2090        require(
2091            _withdrawFeeFactor >= withdrawFeeFactorLL,
2092            "_withdrawFeeFactor too low"
2093        );
2094        require(
2095            _withdrawFeeFactor <= withdrawFeeFactorMax,
2096            "_withdrawFeeFactor too high"
2097        );
2098        withdrawFeeFactor = _withdrawFeeFactor;
2099
2100        require(_controllerFee <= controllerFeeUL, "_controllerFee too high");
2101        controllerFee = _controllerFee;
2102
2103        require(_buyBackRate <= buyBackRateUL, "_buyBackRate too high");
2104        buyBackRate = _buyBackRate;
2105
2106        require(
2107            _slippageFactor <= slippageFactorUL,
2108            "_slippageFactor too high"
2109        );
2110        slippageFactor = _slippageFactor;
2111
2112        emit SetSettings(
2113            _entranceFeeFactor,
2114            _withdrawFeeFactor,
2115            _controllerFee,
2116            _buyBackRate,
2117            _slippageFactor
2118        );
2119    }
```

However, during the contract construction, there is no lower and upper bound limitation to prevent out-of-range value setting to those fee factors. This can lead to incorrect calculation that relies on these fee factor values.

StratX2_PCS.sol

```
2217    contract StratX2_PCS is StratX2 {
2218        constructor(
2219            address[] memory _addresses,
2220            uint256 _pid,
2221            bool _isCAKEStaking,
2222            bool _isSameAssetDeposit,
```

```
2223            bool _isAutoComp,
2224            address[] memory _earnedToAUTOPath,
2225            address[] memory _earnedToToken0Path,
2226            address[] memory _earnedToToken1Path,
2227            address[] memory _token0ToEarnedPath,
2228            address[] memory _token1ToEarnedPath,
2229            uint256 _controllerFee,
2230            uint256 _buyBackRate,
2231            uint256 _entranceFeeFactor,
2232            uint256 _withdrawFeeFactor
2233        ) public {
2234            wbnbAddress = _addresses[0];
2235            govAddress = _addresses[1];
2236            autoFarmAddress = _addresses[2];
2237            AUTOAddress = _addresses[3];
2238
2239            wantAddress = _addresses[4];
2240            token0Address = _addresses[5];
2241            token1Address = _addresses[6];
2242            earnedAddress = _addresses[7];
2243
2244            farmContractAddress = _addresses[8];
2245            pid = _pid;
2246            isCAKEStaking = _isCAKEStaking;
2247            isSameAssetDeposit = _isSameAssetDeposit;
2248            isAutoComp = _isAutoComp;
2249
2250            uniRouterAddress = _addresses[9];
2251            earnedToAUTOPath = _earnedToAUTOPath;
2252            earnedToToken0Path = _earnedToToken0Path;
2253            earnedToToken1Path = _earnedToToken1Path;
2254            token0ToEarnedPath = _token0ToEarnedPath;
2255            token1ToEarnedPath = _token1ToEarnedPath;
2256
2257            controllerFee = _controllerFee;
2258            rewardsAddress = _addresses[10];
2259            buyBackRate = _buyBackRate;
2260            buyBackAddress = _addresses[11];
2261            entranceFeeFactor = _entranceFeeFactor;
2262            withdrawFeeFactor = _withdrawFeeFactor;
2263
2264            transferOwnership(autoFarmAddress);
2265        }
2266    }
```

5.12.2. Remediation

Inspex suggests adding the lower bound and upper bound limitation in the contract constructor to prevent any unexpected value setting using the same criteria as in the `setSettings()` function.

5.13. Unnecessary Function Declaration

ID	IDX-013
Target	Broker
Category	Smart Contract Best Practice
CWE	CWE-1164: Irrelevant Code
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by removing the `_approveTokenForBroker()` function in the `Broker` contract.

5.13.1. Description

The `_approveTokenForBroker()` function is used for checking the current allowance for `simplichef` whether it is sufficient or not. If it is not, it will call the `safeApprove()` to give the allowance to `simplichef`.

By the following function name, it implies that the function will give an allowance of token for the broker, but the following source code instead approves for the `SimpliChef` contract address.

Broker.sol

```
2180 function _approveTokenForBroker(address token) private {
2181     if (IBEP20(token).allowance(address(this), address(simplichef)) == 0) {
2182         IBEP20(token).safeApprove(address(simplichef), type(uint256).max);
2183     }
2184 }
```

However, there is already a function that gives an allowance for the `simplichef`, which is the `_approveTokenForSimpliChef()` function.

Broker.sol

```
39 function _approveTokenForSimpliChef(address token) private {
40     if (IBEP20(token).allowance(address(this), address(simplichef)) == 0) {
41         IBEP20(token).safeApprove(address(simplichef), type(uint256).max);
42     }
43 }
```

Hence, as in the audit scope, the `_approveTokenForBroker()` is not necessary to be declared since there is the `_approveTokenForSimpliChef()` function that does the same job and there is no need to give allowance to itself as implied by the function name.

5.13.2. Remediation

Inspex suggests removing the `_approveTokenForBroker()` function.

5.14. Inexplicit Solidity Compiler Version

ID	IDX-014
Target	Broker SimpliToken ZapBSC
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact SIMPLI FINANCE LAB team has acknowledged this issue.

5.14.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

The affected contract is following table:

Filename	Contract	Version
SimpliToken.sol	SimpliToken	^0.6.12
Broker.sol	Broker	^0.8.0
ZapBSC.sol	ZapBSC	^0.8.0

5.14.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in each major are as follows:

- Major 0.6: v0.6.12
- Major 0.8: v0.8.11

SimpliToken.sol

```
1 // SPDX-License-Identifier: MIT
2
3 pragma solidity 0.6.12;
```

Broker.sol

```
1 pragma solidity 0.8.11;
```

ZapBSC.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.8.11;
```

5.15. Improper Function Visibility

ID	IDX-015
Target	SimpliChef SimpliToken StratX2_PCS ZapBSC
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved SIMPLI FINANCE LAB team has resolved this issue as suggested in commit `4823d1ebbd84a35dd7e8f9fd7f5207ef5c85c419` by applying the new function visibility to all mentioned functions.

5.15.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

The following source code shows that the `setBroker()` function of the `SimpliChef` contract is set to public and it is never called from any internal function.

SimpliChef.sol

```
1492 function setBroker(address _broker) public onlyOwner {
1493     broker = _broker;
1494 }
```

The affected contract is represented in the following table:

Filename	Function
SimpliChef.sol (L: 1492)	setBroker()
SimpliChef.sol (L: 1508)	add()
SimpliChef.sol (L: 1532)	set()
SimpliChef.sol (L: 1641)	deposit()

SimpliChef.sol (L:1672)	depositOnlyBroker()
SimpliChef.sol (L:1799)	withdrawAll()
SimpliChef.sol (L:1804)	emergencyWithdraw()
SimpliChef.sol (L:1831)	inCaseTokensGetStuck()
SimpliToken.sol (L:9)	mint()
StratX2_PCS.sol (L:1785)	deposit()
StratX2_PCS.sol (L:1820)	farm()
StratX2_PCS.sol (L:1820)	earn()
StratX2_PCS.sol (L:2020)	convertDustToEarned()
StratX2_PCS.sol (L:2065)	pause()
StratX2_PCS.sol (L:2069)	unpause()
StratX2_PCS.sol (L:2073)	setSettings()
StratX2_PCS.sol (L:2121)	setGov()
StratX2_PCS.sol (L:2126)	setOnlyGov()
StratX2_PCS.sol (L:2131)	setUniRouterAddress()
StratX2_PCS.sol (L:2140)	setBuyBackAddress()
StratX2_PCS.sol (L:2149)	setRewardsAddress()
StratX2_PCS.sol (L:2158)	inCaseTokensGetStuck()
StratX2_PCS.sol (L:2176)	wrapBNB()
StratX2_PCS.sol (L:2206)	addMember()
StratX2_PCS.sol (L:2211)	removeMember()
Zap_BSC.sol (L:353)	setRoutePairAddress()

5.15.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

SimpliChef.sol

```
1492 function setBroker(address _broker) external onlyOwner {
1493     broker = _broker;
1494 }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] "OWASP Risk Rating Methodology." [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

[2] "TWAP Oracle" [Online]. Available: https://docs.uniswap.org/protocol/V2/concepts/core-concepts/oracles. [Accessed: 01-November-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE