StakingPool, Vault, Strategy & VotingEscrow

Smart Contract Audit Report
Prepared for Scientix

Date Issued:	Sep 9, 2021
Project ID:	AUDIT2021019
Version:	v1.0
Confidentiality Level:	Public

inspex
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021019
Version	v1.0
Client	Scientix
Project	StakingPool, Vault, Strategy & VotingEscrow
Auditor(s)	Weerawat Pawanawiwat Peeraphut Punsuwan
Author	Weerawat Pawanawiwat
Reviewer	Pongsakorn Sommalai
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Sep 9, 2021	Full report	Weerawat Pawanawiwat

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 Scientix, Inspex team conducted an audit to verify the security posture of the StakingPool, Vault, Strategy & VotingEscrow smart contracts between Aug 30, 2021 and Sep 7, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of StakingPool, Vault, Strategy & VotingEscrow 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 5 high, 2 medium, 2 low, 2 very low, and 3 info-severity issues. With the project team's prompt response, 5 high, 2 medium, 1 very low and 1 info-severity issues were resolved or mitigated in the reassessment, while 2 low, 1 very low, and 2 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that StakingPool, Vault, Strategy & VotingEscrow 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

Scientix is a future-yield-backed synthetic asset platform on Binance Smart Chain. It allows the users to get interest-free loans that pay themselves off over time with no liquidation risk, while their collateral earns yields in the background.

StakingPool, Vault, and Strategy are implemented for the users to earn yields on Scientix by depositing their assets to get rewards, and the VotingEscrow allows the users to lock their $SCIX for more $SCIX reward depending on the locking duration.

Scope Information:

Project Name	StakingPool, Vault, Strategy & VotingEscrow
Website	https://scientix.finance/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Aug 30, 2021 - Sep 7, 2021
Reassessment Date	Sep 8, 2021

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:

Contract	Commit	Location (URL)
StakingPools	eda8c5e46e	https://github.com/ScientixFinance/scientix-contract/blob/eda8c5e46e/contracts/StakingPools.sol
SimpleVault	eda8c5e46e	https://github.com/ScientixFinance/scientix-contract/blob/eda8c5e46e/contracts/vaults/SimpleVault.sol
StratAlpaca	eda8c5e46e	https://github.com/ScientixFinance/scientix-contract/blob/eda8c5e46e/contracts/vaults/StratAlpaca.sol
VotingEscrow	234b4e41d6	https://github.com/ScientixFinance/scientix-contract/blob/234b4e41d6/contracts/VotingEscrow.sol

Reassessment:

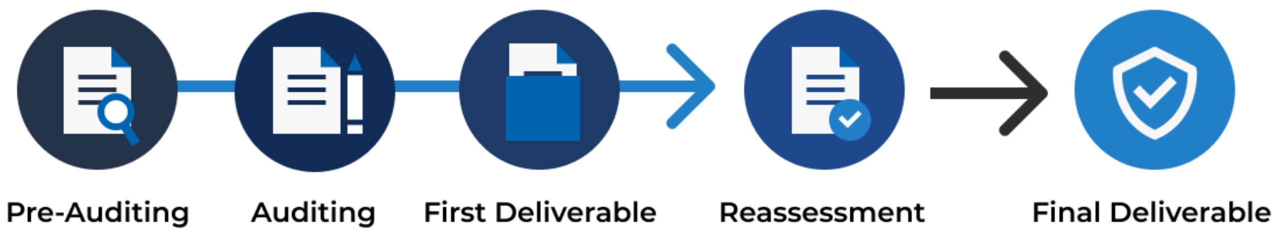
Contract	Commit	Location (URL)
StakingPools	698b8d0226	https://github.com/ScientixFinance/scientix-contract/blob/698b8d0226/contracts/StakingPools.sol
SimpleVault	698b8d0226	https://github.com/ScientixFinance/scientix-contract/blob/698b8d0226/contracts/vaults/SimpleVault.sol
StratAlpaca	698b8d0226	https://github.com/ScientixFinance/scientix-contract/blob/698b8d0226/contracts/vaults/StratAlpaca.sol
VotingEscrow	698b8d0226	https://github.com/ScientixFinance/scientix-contract/blob/698b8d0226/contracts/VotingEscrow.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited 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
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
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

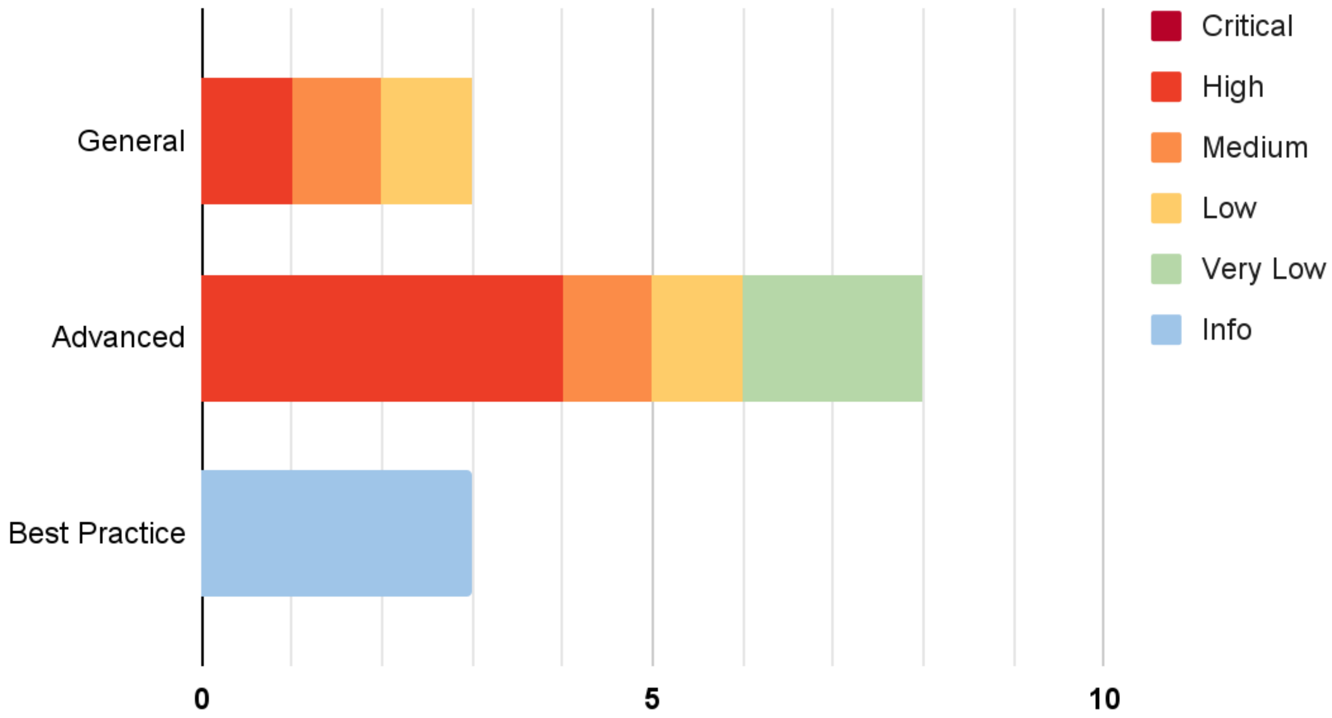
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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**.

Likelihood / Impact	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 14 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	Use of Upgradable Contract Design	Advanced	High	Resolved *
IDX-002	Design Flaw in Contract Initialization	Advanced	High	Resolved
IDX-003	Improper Modification of Strategy Contract Address	Advanced	High	Resolved *
IDX-004	Improper Modification of Vault Contract Address	Advanced	High	Resolved *
IDX-005	Transaction Ordering Dependence	General	High	Resolved *
IDX-006	Centralized Control of State Variable	General	Medium	Resolved *
IDX-007	Improper Reward Configuration Validation	Advanced	Medium	Resolved *
IDX-008	Design Flaw in updatePools() Function	General	Low	Acknowledged
IDX-009	Improper Usage of Upgradable Inheritance	Advanced	Low	Acknowledged
IDX-010	Insufficient Logging for Privileged Functions	Advanced	Very Low	Acknowledged
IDX-011	Improper Lock Time Extension	Advanced	Very Low	Resolved *
IDX-012	Inexplicit Solidity Compiler Version	Best Practice	Info	No Security Impact
IDX-013	Unused Function Parameter	Best Practice	Info	No Security Impact
IDX-014	Improper Function Visibility	Best Practice	Info	Resolved

* The mitigations or clarifications by Scientix can be found in Chapter 5.

5. Detailed Findings Information

5.1. Use of Upgradable Contract Design

ID	IDX-001
Target	SimpleVault StakingPools StratAlpaca VotingEscrow
Category	Advanced 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 user funds anytime they want. **Likelihood: Medium** This action can be performed by the proxy owner without any restriction.
Status	Resolved * The Scientix team has clarified that they will mitigate this issue by setting the contract owner to a timelock and the timelock's admin is a 4/5 multisig DAO. The core signers of the DAO are 2 members from the SCIX team, 2 members from the Alpaca team, and 1 independent co-signer. At the time of the audit, the contracts are not yet deployed, so the ownership of the contracts cannot be verified. The platform users should verify that the timelock is properly used before using or investing on the platform.

5.1.1. 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.

StakingPools changes:

a. Make it to be upgradable
b. Alchemix's emission rate needs to be reduced every epoch (a week) by the DAO governance which seems to be heavy to Scientix team, so Scientix changed it to an automatic way. SCIX (scientix's GOV token)'s emission rate will be 30,000 for the first week, then will be reduced by 500 SCIX every week for 52 weeks, and then the emission rate will be flat from the 53th week (4000 SCIX every week). The details can be found here.

Vault:

a. Code that is written by Scientix team, Alchemix use yearn's Dai vault as its underlying interest generating layer, and Scientix uses Alpaca's BUSD vault but needs a yearn-like layer, so this vault works as Yearn's Dai vault

StratAlpaca

a. Code that is written by Scientix team, Like yearn's vault strategy

From the information provided by the Scientix team, these smart contracts are designed to be upgradable, so the logic of them can be modified by the owner anytime, making the smart contracts untrustworthy.

However, from the current implementation, these smart contracts can be deployed either independently without a proxy contract or deployed as upgradable contracts.

5.1.2. Remediation

Inspex suggests deploying the contracts without the proxy upgrade 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., 3 days. This allows the platform users to monitor the timelock and be notified of the potential changes being done on the smart contracts.

5.2. Design Flaw in Contract Initialization

ID	IDX-002
Target	SimpleVault StakingPools StratAlpaca
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The contract owner can call the `initialize()` function multiple times to replace the critical state variables and steal users' funds or cause unfair changes. **Likelihood:** Medium Only the contract owner can perform this attack; however, there is no restriction to prevent the owner from doing it.
Status	Resolved The Scientix team has resolved this issue in commit `5bde6068b161421722b61ff3e5d06b145414e238` by inheriting OpenZeppelin's `Initializable` contract and use the `initializer` modifier in the constructor of the affected contracts.

5.2.1. Description

The `initialize()` function can be used to initialize the state variables of the contract. If it is called multiple times, the critical state variables that should not be changed will be modified, so it should be limited and used only once. However, there is no restriction to prevent the `initialize()` function from being called multiple times in the affected contracts, for example:

SimpleVault.sol

```
/**
 * @dev Sets the value of {token} to the token that the vault will
 * hold as underlying value. It initializes the vault's own 'moo' token.
 * This token is minted when someone does a deposit. It is burned in order
 * to withdraw the corresponding portion of the underlying assets.
 * @param _token the token to maximize.
 * @param _strategy the address of the strategy.
 * @param _name the name of the vault token.
 * @param _symbol the symbol of the vault token.
 * @param _approvalDelay the delay before a new strat can be approved.
 */
```

```
function initialize(
    address _token,
    address _strategy,
    string memory _name,
    string memory _symbol,
    uint256 _approvalDelay
)
    external
    onlyOwner
{
    assetToken = IERC20(_token);
    strategy = ISimpleStrategy(_strategy);
    approvalDelay = _approvalDelay;
    assetToken.safeApprove(_strategy, uint256(-1));
    vaultName = _name;
    vaultSymbol = _symbol;
}
```

In the example above, the `approvalDelay` state variable is used to delay the changing of the strategy contract. If the `initialize()` function is not restricted, the delay can be set to 0 and the contract owner can instantly drain users' funds from the contract using the issue explained in **IDX-003 Improper Modification of Strategy Contract Address**.

By changing the following critical state variables with the `initialized()` function, the contract owner can drain all users' tokens or cause unfair changes to the users' funds and rewards:

Contract	State Variable
SimpleVault	assetToken
SimpleVault	strategy
SimpleVault	approvalDelay
StakingPools	reward
StakingPools	governance
StakingPools	ctx.rewardRate
StakingPools	ctx.reducedRewardRatePerEpoch
StakingPools	ctx.startBlock
StakingPools	ctx.blocksPerEpoch
StakingPools	ctx.totalReducedEpochs

StratAlpaca	poolId
StratAlpaca	fairLaunch
StratAlpaca	alpacaToken
StratAlpaca	alpacaVault
StratAlpaca	wantToken
StratAlpaca	wbnbAddress
StratAlpaca	uniRouterAddress

5.2.2. Remediation

Inspex suggests verifying that the `initialize()` function must be called only once for all affected contracts.

For example, in the `SimpleVault` contract, a state variable (`initialized`) should be used to check whether the `initialize()` function has been called or not:

SimpleVault.sol

```
bool public needWhitelist = false;
mapping(address => bool) public isWhitelisted;

string private vaultName;
string private vaultSymbol;
bool public initialized;
```

The variable should be used in the `initialize()` function to prevent the `initialize()` function from being called multiple times, for example:

SimpleVault.sol

```
function initialize(
    address _token,
    address _strategy,
    string memory _name,
    string memory _symbol,
    uint256 _approvalDelay
)
    external
    onlyOwner
{
    require(!initialized, "The contract has been initialized");
    initialized = true;
    assetToken = IERC20(_token);
    strategy = ISimpleStrategy(_strategy);
```

```
    approvalDelay = _approvalDelay;
    assetToken.safeApprove(_strategy, uint256(-1));
    vaultName = _name;
    vaultSymbol = _symbol;
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.3. Improper Modification of Strategy Contract Address

ID	IDX-003
Target	SimpleVault
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The contract owner can use the `upgradeStrat()` function to withdraw the users' funds. **Likelihood:** Medium Only the contract owner can call this function; however, there is no restriction to prevent the owner from performing this attack.
Status	Resolved * The Scientix team has clarified that they will mitigate this issue by setting the contract owner to a timelock and the timelock's admin is a 4/5 multisig DAO. The core signers of the DAO are 2 members from the SCIX team, 2 members from the Alpaca team, and 1 independent co-signer. At the time of the audit, the contracts are not yet deployed, so the ownership of the contracts cannot be verified. The platform users should verify that the timelock is properly used before using or investing on the platform.

5.3.1. Description

In the `SimpleVault` contract, the `strategy` state stores the address of the strategy contract used to manage the funds of the users. Whenever the users deposit their tokens to the vault, the tokens are transferred to the strategy contract.

The address of the strategy contract stored can be changed by the contract owner in 2 steps, using the `proposeStrat()` and `upgradeStrat()` functions.

The `proposeStrat()` function sets the address of the new strategy and the time of proposal to the `stratCandidate` variable.

SimpleVault.sol

```
/**
 * @dev Sets the candidate for the new strat to use with this vault.
 * @param _implementation The address of the candidate strategy.
 */
function proposeStrat(address _implementation) external onlyOwner {
    stratCandidate = StratCandidate({
```

```
            implementation: _implementation,
            proposedTime: block.timestamp
        });

        emit NewStratCandidate(_implementation);
    }
```

The `upgradeStrat()` function is then used to apply the change, verifying that a sufficient delay (`approvalDelay`) has passed in line 216, withdrawing the funds from the original strategy contract in line 221, setting the new strategy address from the `stratCandidate` in line 223, and setting the token approval for the new strategy in line 227. This means that in the `strategy.deposit()` function calling in line 228, the new strategy contract can do anything to the funds withdrawn from the original strategy contract.

SimpleVault.sol

```
209 /**
210  * @dev It switches the active strat for the strat candidate. After upgrading, the
211  * candidate implementation is set to the 0x00 address, and proposedTime to a time
212  * happening in +100 years for safety.
213  */
214 function upgradeStrat() external onlyOwner {
215     require(stratCandidate.implementation != address(0), "There is no candidate");
216     require(stratCandidate.proposedTime.add(approvalDelay) < block.timestamp, "Delay has not passed");
217 
218     emit UpgradeStrat(stratCandidate.implementation);
219 
220     strategy.harvest();
221     strategy.withdraw(totalBalance());
222     assetToken.safeApprove(address(strategy), 0);
223     strategy = ISimpleStrategy(stratCandidate.implementation);
224     stratCandidate.implementation = address(0);
225     stratCandidate.proposedTime = 5000000000;
226 
227     assetToken.safeApprove(address(strategy), uint256(-1));
228     strategy.deposit(assetToken.balanceOf(address(this)));
229 }
```

Furthermore, there is no minimum `approvalDelay` in the `initialize()` function, allowing the contract owner to initialize the contract with 0 delay, leading to an unrestricted changing of the `strategy` state.

Therefore, the contract owner can deploy a malicious strategy and use the `upgradeStrat()` function to drain the funds of the contract users with potentially no restriction.

5.3.2. Remediation

In the ideal case, the `strategy` state variable should not be modifiable to keep the integrity of the smart contract.

However, if modifications are needed, Inspex suggests mitigating this issue by setting a sufficient minimum delay for the `approvalDelay` state, e.g., 3 days. This allows the platform users to monitor the changes and be notified of the potential changes being done on the smart contracts.

5.4. Improper Modification of Vault Contract Address

ID	IDX-004
Target	StratAlpaca
Category	Advanced Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The owner can set a new `vault` address and steal users' funds. **Likelihood:** Medium Only the contract owner can perform this attack; however, there is no restriction to prevent the owner from doing it.
Status	Resolved * The Scientix team has clarified that they will mitigate this issue by setting the contract owner to a timelock and the timelock's admin is a 4/5 multisig DAO. The core signers of the DAO are 2 members from the SCIX team, 2 members from the Alpaca team, and 1 independent co-signer. At the time of the audit, the contracts are not yet deployed, so the ownership of the contracts cannot be verified. The platform users should verify that the timelock is properly used before using or investing on the platform.

5.4.1. Description

In the `StratAlpaca` contract, the `vault` state variable is the address of the vault contract that the users can deposit or withdraw tokens from. This variable is used in the `onlyVault` modifier to make sure that only the address in the `vault` state can call the function with this modifier.

StratAlpaca.sol

```
modifier onlyVault() {
    require (msg.sender == vault, "Must from vault");
    _;
}
```

The `withdraw()` function has the `onlyVault` modifier, and can be called by the address in the `vault` state to withdraw the funds from the farming contracts.

StratAlpaca.sol

```
function withdraw(uint256 _wantAmt)
    external
```

```
    override
    onlyVault
    nonReentrantAndUnpaused
{
    uint256 ibAmt = wantAmtToIbAmount(_wantAmt);
    fairLaunch.withdraw(address(this), poolId, ibAmt);
    alpacaVault.withdraw(alpacaVault.balanceOf(address(this)));

    uint256 actualWantAmount = wantToken.balanceOf(address(this));
    wantToken.safeTransfer(
        address(msg.sender),
        actualWantAmount
    );
}
```

The contract owner can set a new `vault` address by using the `setVault()` function to change the `vault` state variable to any address.

StratAlpaca.sol

```
function setVault(address _vault) external onlyOwner {
    vault = _vault;
}
```

Since the contract owner can set the address in the `vault` state to the owner's wallet address, the owner can use that address to call the `withdraw()` function and drain the users' funds from the contract.

5.4.2. Remediation

In the ideal case, the `vault` state variable should not be modifiable to keep the integrity of the smart contract.

However, if modifications are needed, Inspex suggests mitigating this issue by implementing a timelock mechanism with a sufficient delay, e.g., 3 days. This allows the platform users to monitor the changes and be notified of the potential changes being done on the smart contracts.

5.5. Transaction Ordering Dependence

ID	IDX-005
Target	StratAlpaca
Category	General Smart Contract Vulnerability
CWE	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
Risk	**Severity:** High **Impact:** Medium The users will lose a portion of the reward token from the front running attack during the reward compounding. **Likelihood:** High This attack can be done by anyone, and it is profitable if the compounding amount is high, resulting in high motivation for the attack.
Status	Resolved * The Scientix team has mitigated this issue in commit `f027f404a67c51c2b8ef093ced33ed7c50392543` by allowing only the whitelisted keeper addresses to harvest and use `priceMin` parameter to control the price slippage. The team has confirmed that the `priceMin` will be properly set to prevent significant price slippage.

5.5.1. Description

In the `StratAlpaca` contract, the farming reward will be periodically compounded in the `_harvest()` function. This function can be executed by the contract owner through the `harvest()` function, or called whenever the users deposit or withdraw through the `SimpleVault` contract.

This function harvests the reward from the `fairLaunch` contract in line 203 and swaps the reward to the token required by the `alpacaVault` in line 211-217.

StratAlpaca.sol

```
192 function _harvest() internal {
193     if (lastHarvestBlock == block.number) {
194         return;
195     }
196 
197     // Do not harvest if no token is deposited (otherwise, fairLaunch will fail)
198     if (_ibDeposited() == 0) {
199         return;
200     }
201 
```

```
    // Collect alpacaToken
    fairLaunch.harvest(poolId);

    uint256 earnedAlpacaBalance = alpacaToken.balanceOf(address(this));
    if (earnedAlpacaBalance == 0) {
        return;
    }

    if (alpacaToken != wantToken) {
        IPancakeRouter02(uniRouterAddress).swapExactTokensForTokens(
            earnedAlpacaBalance,
            0,
            alpacaToWantPath,
            address(this),
            now.add(600)
        );
    }

    alpacaVault.deposit(IERC20(wantToken).balanceOf(address(this)));
    fairLaunch.deposit(address(this), poolId,
alpacaVault.balanceOf(address(this)));

    lastHarvestBlock = block.number;
}
```

However, on the swapping, the `amountOutMin` parameter of the `swapExactTokensForTokens()` function is set to 0. This means that the price slippage will not be checked, and any amount of resulting token will be accepted, no matter how low it is.

This opens up rooms for the attackers to perform the front-running attack, allowing the attackers to gain profit from the unchecked price tolerance, resulting in a lower reward amount for the platform users.

5.5.2. Remediation

Inspex suggests implementing a price oracle and using the price from the oracle to calculate the acceptable slippage. As an example, TWAP oracle[2] can be used to get the price of the token pair from the on-chain data.

The following example shows how the price from the oracle can be used to calculate the price tolerance (`minAmount`).

StratAlpaca.sol

```
function _harvest() internal {
    if (lastHarvestBlock == block.number) {
        return;
    }
```

```

    // Do not harvest if no token is deposited (otherwise, fairLaunch will fail)
    if (_ibDeposited() == 0) {
        return;
    }

    // Collect alpacaToken
    fairLaunch.harvest(poolId);

    uint256 earnedAlpacaBalance = alpacaToken.balanceOf(address(this));
    if (earnedAlpacaBalance == 0) {
        return;
    }

    if (alpacaToken != wantToken) {
        uint256 oraclePrice = oracle.consult(earnedAlpacaBalance, alpacaToWantPath);
        uint256 minAmount = oraclePrice.mul(PRECISION.sub(SLIPPAGE)).div(PRECISION);
        IPancakeRouter02(uniRouterAddress).swapExactTokensForTokens(
            earnedAlpacaBalance,
            minAmount,
            alpacaToWantPath,
            address(this),
            now.add(600)
        );
    }

    alpacaVault.deposit(IERC20(wantToken).balanceOf(address(this)));
    fairLaunch.deposit(address(this), poolId, alpacaVault.balanceOf(address(this)));

    lastHarvestBlock = block.number;
}
```

5.6. Centralized Control of State Variable

ID	IDX-006
Target	StakingPools StratAlpaca SimpleVault VotingEscrow
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Medium **Impact:** Medium The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done by the owner; however, the changes are limited by fixed values in the smart contracts.
Status	Resolved * The Scientix team has clarified that they will set the contract owner to a timelock and the timelock's admin is a 4/5 multisig DAO. The core signers of the DAO are 2 members from the SCIX team, 2 members from the Alpaca team, and 1 independent co-signer. At the time of the audit, the contracts are not yet deployed, so the ownership of the contracts cannot be verified. The platform users should verify that the timelock is properly used before using or investing on the platform.

5.6.1. 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, as the contract is not yet deployed, there is potentially no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

File	Contract	Function	Modifier
StakingPools.sol (L:112)	StakingPools	initialize()	onlyOwner
StakingPools.sol (L:147)	StakingPools	setPendingGovernance()	onlyGovernance
StakingPools.sol (L:163)	StakingPools	setRewardRate()	onlyGovernance

StakingPools.sol (L:178)	StakingPools	createPool()	onlyGovernance
StakingPools.sol (L:201)	StakingPools	setRewardWeights()	onlyGovernance
SimpleVault.sol (L:66)	SimpleVault	initialize()	onlyOwner
SimpleVault.sol (L:192)	SimpleVault	proposeStrat()	onlyOwner
SimpleVault.sol (L:201)	SimpleVault	whitelist()	onlyOwner
SimpleVault.sol (L:205)	SimpleVault	setNeedWhitelist()	onlyOwner
SimpleVault.sol (L:214)	SimpleVault	upgradeStrat()	onlyOwner
StratAlpaca.sol (L:100)	StratAlpaca	initialize()	onlyOwner
StratAlpaca.sol (L:233)	StratAlpaca	pause()	onlyOwner
StratAlpaca.sol (L:243)	StratAlpaca	unpause()	onlyOwner
StratAlpaca.sol (L:250)	StratAlpaca	setVault()	onlyOwner
VotingEscrow.sol (L:80)	VotingEscrow	initialize()	onlyOwner
VotingEscrow.sol (L:257)	VotingEscrow	depositPoolToken()	onlyOwner
VotingEscrow.sol (L:263)	VotingEscrow	exitPool()	onlyOwner
VotingEscrow.sol (L:267)	VotingEscrow	pause()	onlyOwner
VotingEscrow.sol (L:272)	VotingEscrow	unpause()	onlyOwner
VotingEscrow.sol (L:277)	VotingEscrow	setKeepers()	onlyOwner

5.6.2. 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 sufficient amount of time

5.7. Improper Reward Configuration Validation

ID	IDX-007
Target	StakingPools
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity:** Medium **Impact:** High The contract will be unusable when the reward reduction exceeds the original reward, causing the users to be unable to deposit or withdraw their funds. **Likelihood:** Low It is unlikely for the contract owner to initialize the contract with incorrect values.
Status	Resolved * The Scientix team has clarified that the reward configuration will be set as follows: - `rewardRate` = '145833333333333333'; - `reducedRewardRatePerEpoch` = '2430555555555555'; - `blocksPerEpoch` = 201600; - `totalReducedEpochs` = 52; With this configuration, the reward reduction will not exceed the initial reward rate, and this issue will be mitigated. At the time of the audit, the contracts are not yet deployed, so the reward configuration cannot be verified. The platform users should verify that the reward is properly configured before using or investing on the platform.

5.7.1. Description

In the `StakingPools` contract, the reward emission will be periodically reduced. The initial reward configuration is set at line 129-133 in the `initialize()` function.

StakingPools.sol

```
112 function initialize(
113     IMintableERC20 _reward,
114     address _governance,
115     uint256 _rewardRate,
116     uint256 _reducedRewardRatePerEpoch,
117     uint256 _startBlock,
118     uint256 _blocksPerEpoch,
119     uint256 _totalReducedEpochs
120 )
```

```
    external
    onlyOwner
{
    require(_governance != address(0), "StakingPools: governance address cannot
be 0x0");

    reward = _reward;
    governance = _governance;

    _ctx.rewardRate = _rewardRate;
    _ctx.reducedRewardRatePerEpoch = _reducedRewardRatePerEpoch;
    _ctx.startBlock = _startBlock;
    _ctx.blocksPerEpoch = _blocksPerEpoch;
    _ctx.totalReducedEpochs = _totalReducedEpochs;
}
```

The reward distributed over a period of time is calculated at line 261 with the `update()` function of the `Pool` library.

StakingPools.sol

```
254 /// @dev Claims all rewarded tokens from a pool.
255 ///
256 /// @param _poolId The pool to claim rewards from.
257 ///
258 /// @notice use this function to claim the tokens from a corresponding pool by
    ID.
259 function claim(uint256 _poolId) external nonReentrantAndUnpaused {
260     Pool.Data storage _pool = _pools.get(_poolId);
261     _pool.update(_ctx);
262 
263     Stake.Data storage _stake = _stakes[msg.sender][_poolId];
264     _stake.update(_pool, _ctx);
265 
266     _claim(_poolId);
267 }
```

The `update()` function updates the `accumulatedRewardWeight` state using the `getUpdatedAccumulatedRewardWeight()` function.

Pool.sol

```
/// @dev Updates the pool.
///
/// @param _ctx the pool context.
function update(Data storage _data, Context storage _ctx) internal {
    _data.accumulatedRewardWeight =
_data.getUpdatedAccumulatedRewardWeight(_ctx);
```

```
    _data.lastUpdatedBlock = block.number;
}
```

The `getUpdatedAccumulatedRewardWeight()` function determines the amount of reward to be distributed in the `getBlockReward()` function at line 73.

Pool.sol

```
55 /// @dev Gets the accumulated reward weight of a pool.
56 ///
57 /// @param _ctx the pool context.
58 ///
59 /// @return the accumulated reward weight.
60 function getUpdatedAccumulatedRewardWeight(Data storage _data, Context storage _ctx)
61     internal view
62     returns (FixedPointMath.uq192x64 memory)
63 {
64     if (_data.totalDeposited == 0) {
65         return _data.accumulatedRewardWeight;
66     }
67 
68     uint256 _elapsedTime = block.number.sub(_data.lastUpdatedBlock);
69     if (_elapsedTime == 0) {
70         return _data.accumulatedRewardWeight;
71     }
72 
73     uint256 _distributeAmount = getBlockReward(_ctx, _data.rewardWeight, _data.lastUpdatedBlock, block.number);
74     if (_distributeAmount == 0) {
75         return _data.accumulatedRewardWeight;
76     }
77 
78     FixedPointMath.uq192x64 memory _rewardWeight = FixedPointMath.fromU256(_distributeAmount).div(_data.totalDeposited);
79     return _data.accumulatedRewardWeight.add(_rewardWeight);
80 }
```

The `getBlockReward()` function calculates the current reward rate using the block number, starting block, number of blocks per epoch, and the number of epochs to reduce the reward at line 86 and 93.

Pool.sol

```
82 function getBlockReward(Context memory _ctx, uint256 _rewardWeight, uint256 _from, uint256 _to) internal pure returns (uint256) {
83     uint256 lastReductionBlock = _ctx.startBlock + _ctx.blocksPerEpoch * _ctx.totalReducedEpochs;
84 
```

```
    if (_from >= lastReductionBlock) {
        return _ctx.rewardRate.sub(_ctx.reducedRewardRatePerEpoch.mul(_ctx.totalReducedEpochs))
        .mul(_rewardWeight).div(_ctx.totalRewardWeight)
        .mul(_to - _from);
    }

    uint256 totalRewards = 0;
    if (_to > lastReductionBlock) {
        totalRewards = _ctx.rewardRate.sub(_ctx.reducedRewardRatePerEpoch.mul(_ctx.totalReducedEpochs))
        .mul(_rewardWeight).div(_ctx.totalRewardWeight)
        .mul(_to - lastReductionBlock);

        _to = lastReductionBlock;
    }
    return totalRewards + getReduceBlockReward(_ctx, _rewardWeight, _from, _to);
}
```

The formula for the reward rate after the last reduction is as follows:

```
lastRewardRate = rewardRate - (reducedRewardRatePerEpoch * totalReducedEpochs)
```

And the generic formula for the reward rate in each epoch is as follows:

```
epochRewardRate = rewardRate - (reducedRewardRatePerEpoch * ((currentBlock - startBlock / blocksPerEpoch))
```

As the reward value is calculated using `SafeMath` library, the transaction will revert on integer overflow. The following case must not happen, or else, the reward rate will be in negative, and the transaction with reward calculation will be reverted:

```
_reducedRewardRatePerEpoch * _totalReducedEpochs > rewardRate
```

If the initial reward configuration is not set properly, the contract will be unusable when the reward reduction exceeds the original reward.

5.7.2. Remediation

Inspex suggests verifying the reward configuration in the `initialize()` function by making sure that the reward reduction will not exceed the original reward, for example:

StakingPools.sol

```
function initialize(
    IMintableERC20 _reward,
    address _governance,
    uint256 _rewardRate,
    uint256 _reducedRewardRatePerEpoch,
    uint256 _startBlock,
    uint256 _blocksPerEpoch,
    uint256 _totalReducedEpochs
)
    external
    onlyOwner
{
    require(_governance != address(0), "StakingPools: governance address cannot be 0x0");
    require(_totalReducedEpochs.mul(_reducedRewardRatePerEpoch) <= _rewardRate, "StakingPools: improper reward configuration");
    reward = _reward;
    governance = _governance;

    _ctx.rewardRate = _rewardRate;
    _ctx.reducedRewardRatePerEpoch = _reducedRewardRatePerEpoch;
    _ctx.startBlock = _startBlock;
    _ctx.blocksPerEpoch = _blocksPerEpoch;
    _ctx.totalReducedEpochs = _totalReducedEpochs;
}
```

Please note that the remediations for other issues are not yet applied to the example above.

5.8. Design Flaw in _updatePools() Function

ID	IDX-008
Target	StakingPools.sol
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `_updatePools()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `_pools` size will be raised until the `_updatePools()` function is unusable.
Status	**Acknowledged** The Scientix team has acknowledged this issue because the `createPool()` function is whitelisted for the governance who will not create a lot of invalid pools, so the risks are quite low.

5.8.1. Description

The `_updatePools()` function executes the `pool.update(_ctx)` function, which is a state modifying function for all added pools as shown below:

StakingPools.sol

```
function _updatePools() internal {
  for (uint256 _poolId = 0; _poolId < _pools.length(); _poolId++) {
    Pool.Data storage _pool = _pools.get(_poolId);
    _pool.update(_ctx);
  }
}
```

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

5.8.2. Remediation

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

5.9. Improper Usage of Upgradable Inheritance

ID	IDX-009
Target	SimpleVault StakingPools StratAlpaca
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium A proxy contract that does not inherit the `Ownable` contract cannot be deployed with the affected contracts as its implementation. **Likelihood:** Low It is unlikely for the owner to deploy the contracts with an incompatible proxy contract.
Status	**Acknowledged** The Scientix team has acknowledged this issue because it only happens due to a failed deployment.

5.9.1. Description

The affected contracts, `SimpleVault`, `StakingPools`, and `StratAlpaca`, inherit multiple contracts that are not compatible with the proxy upgrade pattern. The storage initialization is done in the `constructor` of the contract instead of the `initialize()` function commonly used in proxy upgrade pattern, for example, the setting of `owner` in the `constructor` of the `UpgradeableOwnable` contract.

UpgradeableOwnable.sol

```
contract UpgradeableOwnable {
    bytes32 private constant _OWNER_SLOT = 0xa7b53796fd2d99cb1f5ae019b54f9e024446c3d12b483f733ccc62ed04eb126a;

    event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    /**
     * @dev Initializes the contract setting the deployer as the initial owner.
     */
    constructor () internal {
        assert(_OWNER_SLOT == bytes32(uint256(keccak256("eip1967.proxy.owner")) - 1));
        _setOwner(msg.sender);
        emit OwnershipTransferred(address(0), msg.sender);
```

```
    }
```

Furthermore, the affected contracts use the `onlyOwner` modifier in the `initialize()` function, for example:

SimpleVault.sol

```
function initialize(
    address _token,
    address _strategy,
    string memory _name,
    string memory _symbol,
    uint256 _approvalDelay
)
    external
    onlyOwner
{
    assetToken = IERC20(_token);
    strategy = ISimpleStrategy(_strategy);
    approvalDelay = _approvalDelay;
    assetToken.safeApprove(_strategy, uint256(-1));
    vaultName = _name;
    vaultSymbol = _symbol;
}
```

This means that when the affected contracts are deployed, the `owner` will be set in those contracts; however, the owner will not be set when the proxy contract is deployed. If the proxy contract itself does not have the owner at the same storage slot, the deployment of the proxy will fail due to the `onlyOwner` modifier.

The inherited contracts that are not designed for the proxy upgrade pattern are as follows:

Contract	Inherited Contract
StakingPools	UpgradeableOwnable
StakingPools	ReentrancyGuardPausable
SimpleVault	ERC20
SimpleVault	UpgradeableOwnable
SimpleVault	ReentrancyGuardPausable
StratAlpaca	UpgradeableOwnable
StratAlpaca	ReentrancyGuardPausable

5.9.2. Remediation

Inspex suggests inheriting the contracts designed to be used in proxy upgrade pattern[3] for the desired functionalities.

For example, the following contracts from OpenZeppelin can be used instead of the original inherited contracts for the same functionalities:

Inherited Contract	Replacement
UpgradeableOwnable	@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol
ReentrancyGuardPausable	@openzeppelin/contracts-upgradeable/security/ReentrancyGuardUpgradeable.sol and @openzeppelin/contracts-upgradeable/security/PausableUpgradeable.sol
ERC20	@openzeppelin/contracts-upgradeable/token/ERC20/ERC20Upgradeable.sol

For example, the contracts can be inherited as follows:

SimpleVault.sol

```
contract SimpleVault is ERC20Upgradeable, OwnableUpgradeable,
ReentrancyGuardUpgradeable, PausableUpgradeable, IyVaultV2Simple {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;
```

The inherited contracts should also be initialized in the `initialize()` function, for example:

```
function initialize(
    address _token,
    address _strategy,
    string memory _name,
    string memory _symbol,
    uint256 _approvalDelay
)
    external
    initializer
{
    __ERC20_init(_name, _symbol);
    __Ownable_init();
    __ReentrancyGuard_init();
    __Pausable_init();
    assetToken = IERC20(_token);
    strategy = ISimpleStrategy(_strategy);
```

```
    approvalDelay = _approvalDelay;
    assetToken.safeApprove(_strategy, uint256(-1));
    vaultName = _name;
    vaultSymbol = _symbol;
}
```

Please note that the modifiers from the original inherited contracts should be checked and modified accordingly.

5.10. Insufficient Logging for Privileged Functions

ID	IDX-010
Target	StratAlpaca SimpleVault VotingEscrow
Category	Advanced Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	**Acknowledged** The Scientix team has acknowledged this issue.

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 to the platform.

For example, the owner can modify the vault contract address by executing `setVault()` function in the `StratAlpaca` contract, and no event is emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function	Modifier
StratAlpaca.sol (L:233)	StratAlpaca	pause()	onlyOwner
StratAlpaca.sol (L:243)	StratAlpaca	unpause()	onlyOwner
StratAlpaca.sol (L:250)	StratAlpaca	setVault()	onlyOwner
SimpleVault.sol (L:201)	SimpleVault	whitelist()	onlyOwner
SimpleVault.sol (L:205)	SimpleVault	setNeedWhitelist()	onlyOwner
VotingEscrow.sol (L:257)	VotingEscrow	depositPoolToken()	onlyOwner
VotingEscrow.sol (L:263)	VotingEscrow	exitPool()	onlyOwner

VotingEscrow.sol (L:267)	VotingEscrow	pause()	onlyOwner
VotingEscrow.sol (L:272)	VotingEscrow	unpause()	onlyOwner
VotingEscrow.sol (L:294)	VotingEscrow	collectReward()	onlyKeeper

5.10.2. Remediation

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

StratAlpaca.sol

```
event SetVault(address oldVault, address newVault);
function setVault(address _vault) external onlyOwner {
    emit SetVault(vault, _vault);
    vault = _vault;
}
```

5.11. Improper Lock Time Extension

ID	IDX-011
Target	VotingEscrow
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Very Low **Impact:** Low User's token will be locked for a longer duration without receiving any additional reward if the lock duration exceeds `MAX_TIME`. **Likelihood:** Low It is unlikely that the user will extend the total locking duration to be more than the `MAX_TIME`, since there is no benefit for the user.
Status	Resolved * The Scientix team has clarified that they will inform users on how many veSCIX they will get before they extend the lock on the user interface, so the users will be less likely to extend the lock time too much unknowingly.

5.11.1. Description

In the `VotingEscrow` contract, the `claimReward()` modifier calculates the reward and transfers it to the user according to their balance.

VotingEscrow.sol

```
modifier claimReward() {
    _collectReward(false, 0);

    if (_balances[msg.sender] > 0) {
        uint256 pending = _balances[msg.sender].mul(_accRewardPerBalance).div(1e18).sub(_rewardDebt[msg.sender]);
        _scix.safeTransfer(msg.sender, pending);
    }

    _; // _balances[msg.sender] may changed.

    _rewardDebt[msg.sender] = _balances[msg.sender].mul(_accRewardPerBalance).div(1e18);
}
```

The users will be given the most balance when they lock their token with the `MAX_TIME` duration. The lock duration cannot be more than `MAX_TIME` on the lock creation.

VotingEscrow.sol

```
function _createLock(uint256 amount, uint256 end, uint256 timestamp) internal
claimReward {
    LockData storage lock = _locks[msg.sender];

    require(lock.amount == 0, "must no locked");
    require(end <= timestamp + MAX_TIME, "end too long");
    require(end > timestamp, "end too short");
    require(amount != 0, "amount must be non-zero");

    _scix.safeTransferFrom(msg.sender, address(this), amount);
    _totalLockedSCIX = _totalLockedSCIX.add(amount);

    lock.amount = amount;
    lock.end = end;

    _updateBalance(msg.sender, (end - timestamp).mul(amount).div(MAX_TIME));

    emit LockCreate(msg.sender, lock.amount, _balances[msg.sender], lock.end);
}
```

The user can extend the lock duration to a new point of time by setting the new ending point of the lock. However, the duration can exceed the `MAX_TIME`, for example, the user can start locking their token for `MAX_TIME` (1460 days/4 years), and after 2 years, the user can extend the lock time for another 4 years, making the total duration to be 6 years

In line 228-230, if the duration exceeds `MAX_TIME`, the duration will be set to `MAX_TIME`, and there will be no additional reward for the user.

VotingEscrow.sol

```
219 function _extendLock(uint256 end, uint256 timestamp) internal claimReward {
220     LockData storage lock = _locks[msg.sender];
221     require(lock.amount != 0, "must locked");
222     require(lock.end < end, "new end must be longer");
223     require(end <= timestamp + MAX_TIME, "end too long");
224
225     // calculate equivalent lock duration
226     uint256 duration = _balances[msg.sender].mul(MAX_TIME).div(lock.amount);
227     duration += (end - lock.end);
228     if (duration > MAX_TIME) {
229         duration = MAX_TIME;
230     }
231
```

```
    lock.end = end;
    _updateBalance(msg.sender, duration.mul(lock.amount).div(MAX_TIME));

    emit LockExtend(msg.sender, lock.amount, _balances[msg.sender], lock.end);
}
```

This can be bad for the platform users, since they may unknowingly extend their lock time exceeding the `MAX_TIME`, and won't be able to withdraw for a longer time without any additional benefit.

5.11.2. Remediation

Inspex suggests reverting the lock extension transaction if the total `duration` is more than `MAX_TIME` since the user will not get any further benefit from exceeding `MAX_TIME`, for example:

VotingEscrow.sol

```
function _extendLock(uint256 end, uint256 timestamp) internal claimReward {
    LockData storage lock = _locks[msg.sender];
    require(lock.amount != 0, "must locked");
    require(lock.end < end, "new end must be longer");
    require(end <= timestamp + MAX_TIME, "end too long");

    // calculate equivalent lock duration
    uint256 duration = _balances[msg.sender].mul(MAX_TIME).div(lock.amount);
    duration += (end - lock.end);
    require(duration <= MAX_TIME, "duration too long");

    lock.end = end;
    _updateBalance(msg.sender, duration.mul(lock.amount).div(MAX_TIME));

    emit LockExtend(msg.sender, lock.amount, _balances[msg.sender], lock.end);
}
```

5.12. Inexplicit Solidity Compiler Version

ID	IDX-012
Target	StakingPools StratAlpaca SimpleVault
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 The Scientix team has acknowledged this issue.

5.12.1. 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, for example:

StakingPools.sol

```
pragma solidity ^0.6.12;
//SPDX-License-Identifier: MIT
```

The following table contains all targets which the inexplicit compiler version is declared.

Contract	Version
StakingPools	^0.6.12
StratAlpaca	^0.6.0
SimpleVault	^0.6.0
VotingEscrow	>=0.6.0 <0.8.0

5.12.2. 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 major 0.6 is v0.6.12.

5.13. Unused Function Parameter

ID	IDX-013
Target	StratAlpaca VotingEscrow
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 The Scientix team has acknowledged this issue.

5.13.1. Description

There are parameters defined in multiple functions of the smart contracts that are not used anywhere, causing unnecessary gas usage.

For example, In the `StratAlpaca` contract, the `_flags` parameter in `pause()` and `unpause()` functions are declared but not used anywhere in the functions.

StratAlpaca.sol

```
/**
 * @dev Pauses the strat.
 */
function pause(uint256 _flags) external onlyOwner {
    _pause();

    alpacaToken.safeApprove(uniRouterAddress, 0);
    wantToken.safeApprove(uniRouterAddress, 0);
}

/**
 * @dev Unpauses the strat.
 */
function unpause(uint256 _flags) external onlyOwner {
    _unpause();

    alpacaToken.safeApprove(uniRouterAddress, uint256(-1));
    wantToken.safeApprove(uniRouterAddress, uint256(-1));
}
```

The unused function parameters are as follows:

File	Contract	Function	Parameter
StratAlpaca.sol (L:239)	StratAlpaca	pause()	flags
StratAlpaca.sol (L:249)	StratAlpaca	unpause()	flags
VotingEscrow.sol (L:267)	VotingEscrow	pause()	flags
VotingEscrow.sol (L:272)	VotingEscrow	unpause()	flags

5.13.2. Remediation

Inspex suggests removing the unused parameter to reduce unnecessary gas usage.

5.14. Improper Function Visibility

ID	IDX-014
Target	VotingEscrow
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The Scientix team has resolved this issue as suggested in commit 698b8d0226fef814a9bdea9451547400c967234a.

5.14.1. 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.

For example, the following source code shows that the `collectReward()` function of the `VotingEscrow` contract is set to public and it is never called from any internal function.

VotingEscrow.sol

```
294 function collectReward(bool buyback) public onlyKeeper nonReentrantAndUnpaused
    {
295     _collectReward(buyback);
296 }
```

The following table contains all functions that have `public` visibility and are never called from any internal function.

File	Contract	Function
VotingEscrow.sol (L:142)	VotingEscrow	approve()
VotingEscrow.sol (L:146)	VotingEscrow	transferFrom()
VotingEscrow.sol (L:294)	VotingEscrow	collectReward()

5.14.2. Remediation

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

VotingEscrow.sol

```
function collectReward(bool buyback) external onlyKeeper
nonReentrantAndUnpaused {
    _collectReward(buyback);
}
```

6. Appendix

6.1. About Inspex

CYBERSECURITY PROFESSIONAL SERVICE

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] "Oracles | Uniswap." [Online]. Available: https://docs.uniswap.org/protocol/V2/concepts/core-concepts/oracles. [Accessed: 09-Sep-2021]

[3] "Proxy Upgrade Pattern." [Online]. Available: https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies. [Accessed: 09-Sep-2021]

inspex

CYBERSECURITY PROFESSIONAL SERVICE