Security Audit Report

Scientix Protocol

SECBIT

September 7, 2021

1. Introduction

Scientix Protocol is a future-yield-backed synthetic asset platform on Binance Smart Chain. SECBIT Labs conducted an audit from August 28th to September 7th, 2021, including an analysis of the contract in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Scientix Protocol contract has no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising(see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 Restrict the `initialize()` function to allow it to be called only once.	Info	Fixed
Design & Implementation	4.3.2 Adjusting the order of the arithmetic operations can improve the accuracy of the calculation results while maintaining safety.	Info	Fixed
Design & Implementation	4.3.3 A potential sandwich attack could hijack all the earnings of the project.	Low	Fixed
Gas Optimization	4.3.4 Two function interfaces that repeatedly implement the same behavior.	Info	Fixed
Discussion	4.3.5 It will consume a large amount of the user's gas to retrieve reward by calling the `harvest()` function frequently.	Info	Fixed
Design & Implementation	4.3.6 Add stricter constraints on core functions to prevent flash loan attacks.	Info	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about the Scientix Protocol contract is shown below:

- Project website
  - https://scientix.finance/
- Smart contract code
  - https://github.com/ScientixFinance/scientix-contract
  - initial review commit *eda8c5e*
  - final review commit *234b4e4*

2.2 Contract List

The following content shows the contracts included in the Scientix Protocol project, which the SECBIT team audits:

Name	Lines	Description
TransmuterB.sol	441	A contract that allows users to stake to redeem farm tokens.
StakingPools.sol	233	A contract that allows users to stake to farm tokens.
Pool.sol	107	A library that provides the Pool data struct and associated functions.
Stake.sol	38	A library that provides the Stake data struct and associated functions.
SimpleVault.sol	134	A contract that implements a vault to deposit funds for optimizing yield.
StratAlpaca.sol	177	A strategy contract is used to handle the farming and harvesting of user funds.
VotingEscrow	249	A contract for locking SCIX token, and users can get earnings.

Note: The Scientix Protocol is forked from https://github.com/alchemix-finance/alchemix-protocol. Only minor changes have been made to the underlying core code. This audit focused on the changed code as well as the files mentioned above.

3. Contract Analysis

This part describes code assessment details, including two items: "role classification" and "functional analysis".

3.1 Role Classification

There are two key roles in the protocol, namely Governance Account and Common Account.

- Governance Account
    - Description Contract administrator
    - Authority
        - Update basic parameters
        - Create staking pool
        - Manage user funds
        - Adjust contract strategies
    - Method of Authorization The contract administrator is the creator of the contract or authorized by the transferring of governance account.
- Common Account
    - Description Users participate in Scientix Protocol
    - Authority
        - deposit the specified token into the `StakingPool` to receive the earnings
        - deposit funds into the vault
    - Method of Authorization No authorization required

3.2 Functional Analysis

The Scientix Protocol provides users with advances on yield farming through a synthetic token representing a fungible claim on any underlying collateral. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into several parts:

StakingPools

Users will be rewarded with SCIX tokens by depositing specified token into the pool of this contract. `Stake` contract and `Pool` contract are both auxiliary contracts to this contract.

The main functions in `StakingPools` are as below:

- `createPool()`

  The contract administrator uses this function to create a stake pool and also to set the initialization parameters. These stake pools have different weights. Users can obtain the corresponding amount of SCIX tokens by depositing specified funds into different pools.

- `deposit()`

  This function allows the user to deposit the specified number of tokens into the specified pool.

- `withdraw()`

  A User withdraws the specified amount of principal and all rewards under the specified pool.

- `claim()`

  A User retrieves the reward under the specified pool.

- `exit()`

  A User claims all rewards from a pool and then withdraws all staked tokens.

SimpleVault

The vault deposits BUSD into ALPACA protocol to get the lending interests and ibBUSD stake rewards.

The main functions in `SimpleVault` are as below:

- `deposit()`

  This function allows the user to deposit BUSD tokens into the vault and obtain the corresponding shares.

- `withdraw()`

  The user burns a specified number of shares and gets back the principal and the corresponding earnings.

StratAlpaca

This contract transfers the funds deposited in the vault to the Alpaca protocol and receives the proceeds.

The main functions in `StratAlpaca` are as below:

- `deposit()`

  Only vault contract is allowed to call this function. The function is used to transfer the BUSD token deposited under the vault contract to the Alpaca protocol.

- `withdraw()`

  Only vault contract is allowed to call this function. In contrast to the `deposit()` function, this function retrieves the funds deposited under the Alpaca protocol.

- `harvest()`

  This function retrieves the reward Alpaca token under the Alpaca protocol and swaps it into BUSD token for reinvestment.

VotingEscrow

User deposits their SCIX token into the contract and specifies the duration of the deposit. Based on the length of deposit (maximum four years) and the number of funds deposited, the share of the user is calculated. The longer the stake time, the larger the share and the more revenue user will receive.

The main functions in `VotingEscrow` are as below:

- `createLock()`

  The user deposits the SCI token and specifies the time to withdraw the funds.

- `addAmount()`

  This function allows the user to deposit additional funds, at which point the user's share of funds is updated.

- `extendLock()`

  This function allows the user to extend the deposit time and get more earnings.

- `withdraw()`

  After the deposit time expires, the user calls this function to retrieve the principal deposited.

- `claim()`

  The user calls this function to retrieve the revenue in real-time.

4. Audit Detail

This part describes the process, and the detailed results of the audit also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bug, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into two types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓

4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20 standard	✓
7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓

21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 Restrict the `initialize()` function to allow it to be called only once.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Implementation logic	Fixed

Description

When using the proxy upgrade pattern, the logic contract should move the code within the `constructor()` function to a regular `initialize()` function. This function should be called whenever the proxy links to this logic contract. Special care needs to be taken with this `initialize()` function so that it can only be called once, which is one of the properties of constructors in general programming. This is why when we create a proxy using OpenZeppelin Upgrades, we can provide the name of the `initialize()` function and pass parameters.

```
// @audit located on StakingPools.sol
contract StakingPools is UpgradeableOwnable,
ReentrancyGuardPausable {
    ......
    function initialize(
    IMintableERC20 _reward,
    address _governance,
    uint256 _rewardRate,
    uint256 _reducedRewardRatePerEpoch,
    uint256 _startBlock,
    uint256 _blocksPerEpoch,
    uint256 _totalReducedEpochs
```

```
    )
      external
      onlyOwner
    {
        ......
    }
    ......
}
```

Suggestion

A simple modifier is used to ensure that the `initialize()` function can only be called once. OpenZeppelin Upgrades provides this functionality via a contract that can be extended, see: Initializable.sol .

The corresponding modifications are as follows:

```
import "https://github.com/OpenZeppelin/openzeppelin-
contracts-
upgradeable/blob/master/contracts/proxy/utils/Initializable.so
l";

contract StakingPools is UpgradeableOwnable,
                         ReentrancyGuardPausable,
                         Initializable {
    ......
    function initialize(
    IMintableERC20 _reward,
    address _governance,
    uint256 _rewardRate,
    uint256 _reducedRewardRatePerEpoch,
    uint256 _startBlock,
    uint256 _blocksPerEpoch,
    uint256 _totalReducedEpochs
    )
      external
      initializer
    {
        ......
```

```
    }
    ......
}
```

As `SimpleVault.sol` contract and `StratAapaca.sol` contract also use OpenZeppelin Upgrade pattern, their `initialize()` functions should be modified as same as above.

Status

The development team adopts our advice and fix this issue in commit 5bde606.

4.3.2 Adjusting the order of the arithmetic operations can improve the accuracy of the calculation results while maintaining safety.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Implementation logic	Fixed

Description

In cases where the result of an arithmetic operation does not overflow, the order of the operations should be adjusted. For example, multiplying first and dividing last will improve the accuracy of the result.

```
// @audit located on Pool.sol
function getBlockReward(Context memory _ctx, uint256
_rewardWeight, uint256 _from, uint256 _to) internal pure
returns (uint256) {
    ......
    if (_from >= lastReductionBlock) {
      return
_ctx.rewardRate.sub(_ctx.reducedRewardRatePerEpoch.mul(_ctx.to
talReducedEpochs))
      .mul(_rewardWeight).div(_ctx.totalRewardWeight)
      .mul(_to - _from);
    }
```

```
    ......
    if (_to > lastReductionBlock) {
      totalRewards =
_ctx.rewardRate.sub(_ctx.reducedRewardRatePerEpoch.mul(_ctx.to
talReducedEpochs))
      .mul(_rewardWeight).div(_ctx.totalRewardWeight)
      .mul(_to - lastReductionBlock);

      .....
    }
    ......
  }


function getReduceBlockReward(Context memory _ctx, uint256
_rewardWeight, uint256 _from, uint256 _to) internal pure
returns (uint256) {
      ......
      if (right > left) {
        totalRewards +=
rewardPerBlock.mul(_rewardWeight).div(_ctx.totalRewardWeight).
mul(right - left);
      }

      ......
  }
```

Suggestion

The order of calculation can be adjusted. A recommended modification is as follows:

```
function getBlockReward(Context memory _ctx, uint256
_rewardWeight, uint256 _from, uint256 _to) internal pure
returns (uint256) {
    ......
    if (_from >= lastReductionBlock) {
```

```
      return
_ctx.rewardRate.sub(_ctx.reducedRewardRatePerEpoch.mul(_ctx.to
talReducedEpochs))
      .mul(_rewardWeight).mul(_to - _from)
      .div(_ctx.totalRewardWeight);
    }

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

      .....
    }
    ......
  }


function getReduceBlockReward(Context memory _ctx, uint256
_rewardWeight, uint256 _from, uint256 _to) internal pure
returns (uint256) {
      ......
      if (right > left) {
        totalRewards +=
rewardPerBlock.mul(_rewardWeight).mul(right -
left).div(_ctx.totalRewardWeight);
      }

      ......
  }
```

Status

The development team adopts our advice and fix this issue in commit 5bde606.

4.3.3 A potential sandwich attack could hijack all the earnings of the project.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Low	Implementation logic	Fixed

Description

The `harvest()` function is triggered when the user calls `deposit()` function or `withdraw()` function. This function retrieves the earnings and exchanges them into BUSD tokens for reinvestment by the pancakeswap pool. Slippage protection is not in place when using pancakeswap for token exchange, resulting in a possible sandwich attack.

Specifically, in this contract, the user swaps Alpaca token rewards for the WBNB token. The WBNB token is then exchanged for the BUSD token. There is no limit to the minimum number of tokens that can be swapped, which means that the number of tokens the user gets in exchange could be zero. An attacker could easily do this by controlling the exchange ratio of the pancakeswap pool.

```
// @audit located on StratAlpaca.sol
function _harvest() internal {
        ......
        if (alpacaToken != wantToken) {

 IPancakeRouter02(uniRouterAddress).swapExactTokensForTokens(
                earnedAlpacaBalance,
                0,
                alpacaToWantPath,
                address(this),
                now.add(600)
```

```
            );
        }

        ......
    }

function harvest() external override onlyVaultOrOwner
nonReentrantAndUnpaused {
        _harvest();
    }
```

Suggestion

One recommended method of fixing this is to separate the `harvest()` function from `deposit()` and `withdraw()` operations. And only one special role is allowed to execute the function.

Status

The developer team solved this problem by adding function call restriction in commit 5bde606.

4.3.4 Two function interfaces that repeatedly implement the same behavior.

Risk Type	Risk Level	Impact	Status
Gas Optimization	Info	More gas consumption	Fixed

Description

The `getPricePerFullShare()` function and the `pricePerShare()` function return the same value. It will cause more gas to be consumed when the contract is deployed.

```
// @audit located on SimpleVault.sol
function getPricePerFullShare() public view returns (uint256)
{
        return totalBalance().mul(1e18).div(totalSupply());
      }

function pricePerShare() public view override returns
(uint256) {
        return getPricePerFullShare();
      }
```

Status

This issue has been discussed.

4.3.5 It will consume a large amount of the user's gas to retrieve reward by calling the `harvest()` function frequently.

Risk Type	Risk Level	Impact	Status
Discussion	Info	Implementation logic	Fixed

Description

In the `SimpleVault` contract, the `deposit()` function and the `withdraw()` function are high frequency trading operations. Both functions will indirectly call the `harvest()` function.

The `harvest()` function retrieves the reward of the `SimpleVault` contract account for reinvestment. The `harvest()` function is called frequently so that the value of the harvest is small each time. However, this design consumes a lot of the user's gas and raises the cost of participating in the project. It is necessary to consider adjusting the frequency of calls to the `harvest()` function.

```
// @audit located on SimpleVault.sol
```

```
function deposit(uint256 _amount) public override
nonReentrantAndUnpaused returns (uint256) {
        ......
        strategy.harvest();

        ......
    }

function _withdraw(uint256 _shares, address _recipient)
internal returns (uint256) {
        strategy.harvest();

        ......
    }

// @audit located on 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) {

 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;
    }

function harvest() external override onlyVaultOrOwner
nonReentrantAndUnpaused {
        _harvest();
    }
```

Status

This issue has been discussed.

4.3.6 Add stricter constraints on core functions to prevent flash loan attacks.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Implementation logic	Fixed

Description

For core functions in the contract that involve the transfer of funds, the security of the function should be enhanced to prevent possible flash loan attacks. Some of the core functions in the contract are as follows:

```
// @audit located on TransmuterB.sol
function unstake(uint256 amount) public
updateAccount(msg.sender) {
        ......
    }

function stake(uint256 amount)
        public
        runPhasedDistribution()
        updateAccount(msg.sender)
        checkIfNewUser()
    {
       ......
    }

function transmute() public runPhasedDistribution()
updateAccount(msg.sender) {
       ......
}

function forceTransmute(address toTransmute)
        public
        runPhasedDistribution()
        updateAccount(msg.sender)
        updateAccount(toTransmute)
        checkIfNewUser()
    {
        ......
    }
```

Suggestion

There are two methods commonly used to prevent flash loan attacks: function disabling contract calls and function execution delay.

Status

The development team adopts our advice and fix this issue in commit 5bde606.

5. Conclusion

After auditing and analyzing the Scientix Protocol contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above. SECBIT Labs holds the view that the Scientix Protocol contract has high code quality, concise implementation, and detailed documentation.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock ethers inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract, could possibly bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

https://secbit.io

audit@secbit.io

@secbit_io