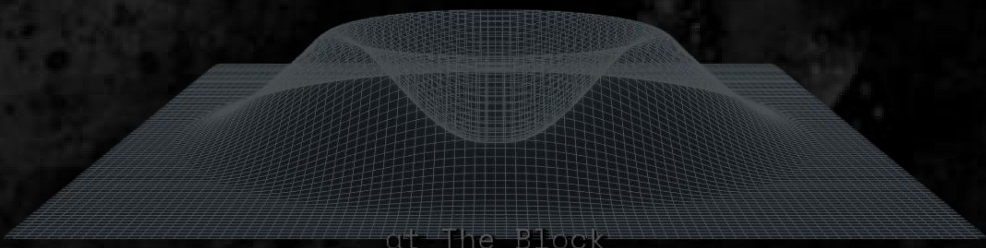
CHAINLION

ULME （TRADING POOL）

Smart Contract Audit Report

NOV 07th，2022

NO.0C002211070004



CATALOGUE

1. PROJECT SUMMARY

Entry type	Specific description
Entry name	ULME TRADING POOL
Project type	**ERC-20**
Application platform	BSC
DawnToken	0xf18e5ec98541d073daa0864232b9398fa183e0d4

2. AUDIT SUMMARY

Entry type	Specific description
Project cycle	NOV/02/2022-NOV/07/2022
Audit method	Black box test、White box test、Grey box test
Auditors	TWO

3. VULNERABILITY SUMMARY

Audit results are as follows:

Entry type	Specific description
Serious vulnerability	0
High risk vulnerability	0
Moderate risk	0

Low risk vulnerability	0

Security vulnerability rating description：

1) **Serious vulnerability：** Security vulnerabilities that can directly cause token contracts or user capital losses， For example: shaping overflow vulnerability、Fake recharge vulnerability、Reentry attacks, vulnerabilities, etc.

2) **High risk vulnerability：** Security vulnerabilities that can directly cause the contract to fail to work normally, such as reconstructed smart contract caused by constructor design error, denial of service vulnerability caused by unreasonable design of require / assert detection conditions, etc.

3) **Moderate risk：** Security problems caused by unreasonable business logic design, such as accuracy problems caused by unreasonable numerical operation sequence design, variable ambiguous naming, variable coverage, call injection, conditional competition, etc.

4) **Low risk vulnerability：** Security vulnerabilities that can only be triggered by users with special permissions, such as contract backdoor vulnerability, duplicate name pool addition vulnerability, non-standard contract coding, contract detection bypass, lack of necessary events for key state variable change, and security vulnerabilities that are harmful in theory but have harsh utilization conditions.

4. EXECUTIVE SUMMARY

This report is prepared for **ULME** TRADING POOL smart contract，The purpose is to find the security vulnerabilities and non-standard coding problems in the smart contract through the security audit of the source code of the smart contract. This audit mainly involves the following test methods:

White box test

Conduct security audit on the source code of smart contract and check the security issues such as coding specification, DASP top 10 and business logic design

Grey box test

Deploy smart contracts locally and conduct fuzzy testing to check function robustness, function call permission and business logic security

Black box test

Conduct security test attacks on smart contracts from the perspective of attackers, combined with black-and-white and testing techniques, to check whether there are exploitable vulnerabilities.

This audit report is subject to the latest contract code provided by the current project party, does not include the newly added business logic function module after the contract upgrade, does not include new attack methods in the future, and does not include web front-end security and server-side security.

5. Directory structure

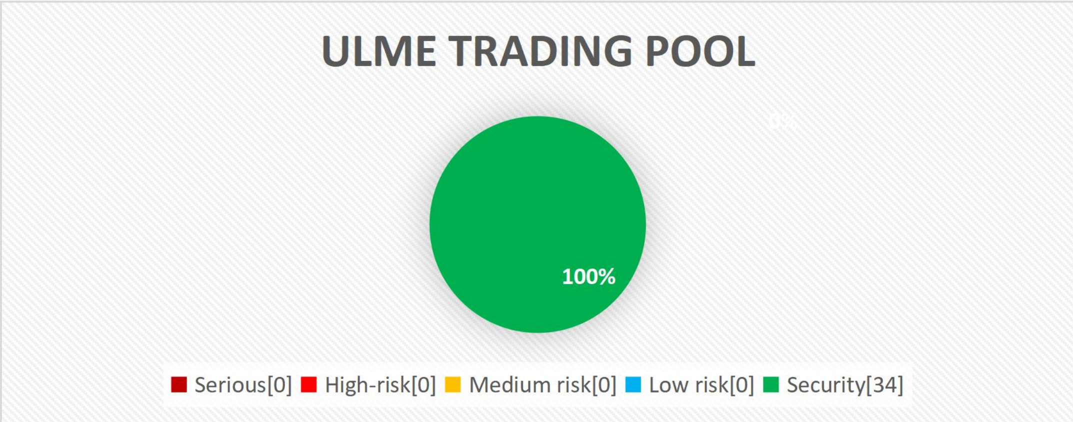
```
PancakePair.sol
```

6. File hashes

Contract	SHA1 Checksum

PancakePair.sol	B626327449A35543EAF3DAFE4C573EA50C7EA505

7. Vulnerability distribution



8. Audit content

8.1. Coding specification

Smart contract supports contract development in programming languages such as solid, Vyper, C + +, Python and rust. Each programming language has its own coding specification. In the development process, the coding specification of the development language should be strictly followed to avoid security problems such as business function design defects.

8.1.1. Compiler Version【security】

Audit description： The compiler version should be specified in the smart contract code. At the same time, it is recommended to use the latest compiler version. The old version of the compiler may cause various known security problems. At present, the latest version is v 0.8 x. And this version has been protected against shaping overflow.

Audit results： According to the audit, the compiler version used in the smart contract code is 0.8.0, so there is no such security problem.

```
/**
 *Submitted for verification at BscScan.com on 2021-04-23
*/

// File: contracts\interfaces\IPancakePair.sol

pragma solidity >=0.5.0;

interface IPancakePair {
    event Approval(address indexed owner, address indexed spender, uint value);
    event Transfer(address indexed from, address indexed to, uint value);

    function name() external pure returns (string memory);
    function symbol() external pure returns (string memory);
    function decimals() external pure returns (uint8);
    function totalSupply() external view returns (uint);
    function balanceOf(address owner) external view returns (uint);
    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint value) external returns (bool);
    function transfer(address to, uint value) external returns (bool);
    function transferFrom(address from, address to, uint value) external returns (bool);

    function DOMAIN_SEPARATOR() external view returns (bytes32);
    function PERMIT_TYPEHASH() external pure returns (bytes32);
    function nonces(address owner) external view returns (uint);
```

Safety advice： NONE.

8.1.2. Return value verification【security】

Audit description： Smart contract requires contract developers to strictly follow EIP / tip and other standards and specifications during contract development. For transfer, transferfrom and approve functions, Boolean values should be returned to feed back the final execution results. In the smart contract, the relevant business logic code often calls the transfer or transferfrom function to transfer. In this case, the return value involved in the transfer operation should be strictly checked to determine whether the transfer is successful or not, so as to avoid security vulnerabilities such as false recharge caused by the lack of return value verification.

Audit results： According to the audit, there is no embedded function calling the official standards transfer and transferfrom in the smart contract, so there is no such security problem.

Safety advice： NONE.

8.1.3. Constructor writing【security】

Audit description： In solid v0 The smart contract written by solidity before version 4.22 requires that the constructor must be consistent with the contract name. When the constructor name is inconsistent with the contract name, the constructor will become an ordinary public function. Any user can call the constructor to initialize the contract. After version V 0.4.22, The constructor name can be replaced by constructor, so as to avoid the coding problems caused by constructor writing.

Audit results： After audit, the constructor in the smart contract is written correctly, and there is no such security problem.

```
    constructor() public {
        factory = msg.sender;
    }
```

Safety advice： NONE.

8.1.4. Key event trigger【security】

Audit description： Most of the key global variable initialization or update operations similar to setXXX exist in the smart contract. It is recommended to trigger the corresponding event through emit when operating on similar key events.

Audit results： No relevant security risk is found through audit.

Safety advice： NONE.

8.1.5. Address non-zero check【security】

Audit description： The smart contract initializes the key information of the contract through the constructor. When it comes to address initialization, the address should be non-zero checked to avoid irreparable economic losses.

Audit results： No relevant security risk is found through audit.

Safety advice： NONE.

8.1.6. Code redundancy check【security】

Audit description： The deployment and execution of smart contracts need to consume certain gas costs. The business logic design should be optimized as much as possible, while avoiding unnecessary redundant code to improve efficiency and save costs.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2. Coding design

DASP top 10 summarizes the common security vulnerabilities of smart contracts. Smart contract developers can study smart contract security vulnerabilities before developing contracts to avoid security vulnerabilities during contract development. Contract auditors can quickly audit and check the existing security vulnerabilities of smart contracts according to DASP top 10.

8.2.1. Shaping overflow detection【security】

Audit description： Solid can handle 256 digits at most. When the number is unsigned, the maximum value will overflow by 1 to get 0, and 0 minus 1 will overflow to get the maximum value. The problem of shaping overflow often appears in the relevant logic code design function modules such as transaction transfer, reward calculation and expense calculation. The security problems caused by shaping overflow are also very serious, such as excessive coinage, high sales and low income, excessive distribution, etc. the problem of shaping overflow can be solved by using solid V 0.8 X version or by using the safemath library officially provided by openzenppelin.

Audit results： According to the audit, the smart contract uses the SafeMath function library built by the project party to conduct numerical operations, which can better prevent the shaping overflow problem caused by numerical operations.

```
// a library for performing overflow-safe math, courtesy of DappHub (https://github.com/dapphub/ds-math)

library SafeMath {
    function add(uint x, uint y) internal pure returns (uint z) {
        require((z = x + y) >= x, 'ds-math-add-overflow');
    }

    function sub(uint x, uint y) internal pure returns (uint z) {
        require((z = x - y) <= x, 'ds-math-sub-underflow');
    }

    function mul(uint x, uint y) internal pure returns (uint z) {
        require(y == 0 || (z = x * y) / y == x, 'ds-math-mul-overflow');
    }
}
```

Safety advice：NONE.

8.2.2. Reentry detection【security】

Audit description： The in solidity provides call Value(), send(), transfer() and other functions are used for transfer operation. When call When value() sends ether, it will send all gas for transfer operation by default. If the transfer function can be called recursively again through call transfer, it can cause reentry attack.

Audit results： It is audited that there is a relevant lock mechanism to prevent reentry attacks.

```
uint private unlocked = 1;
    modifier lock() { //Lock mechanism to prevent reentry attacks
        require(unlocked == 1, 'Pancake: LOCKED');
        unlocked = 0;
        _;
        unlocked = 1;
    }
```

Safety advice： NONE.

8.2.3. Rearrangement attack detection 【security】

Audit description：Rearrangement attack means that miners or other parties try to compete with smart contract participants by inserting their information into the list or mapping, so that attackers have the opportunity to store their information in the contract.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.4. Replay Attack Detection 【security】

Audit description： When the contract involves the business logic of delegated management, attention should be paid to the non reusability of verification to avoid replay attacks. In common asset management systems, there are often delegated management businesses. The principal gives the assets to the trustee for management, and the principal pays a certain fee to the trustee. In similar delegated management scenarios, it is necessary to ensure that the verification information will become invalid once used.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.5. False recharge detection 【security】

Audit description：When a smart contract uses the transfer function for transfer, it should use require / assert to strictly check the transfer conditions. It is not recommended to use if Use mild judgment methods such as else to check, otherwise it will misjudge the success of the transaction, resulting in the security problem of false recharge.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.6. Access control detection 【security】

Audit description：Solid provides four function access domain Keywords: public, private, external and internal to limit the scope of function. In the smart contract, the scope of function should be reasonably designed to avoid the security risk of improper access control. The main differences of the above four keywords are as follows:

1．public： The marked function or variable can be called or obtained by any account, which can be a function in the contract, an external user or inherit the function in the contract

2．external：The marked functions can only be accessed from the outside and cannot be called directly by the functions in the contract, but this can be used Func() calls this function as an external call

3．private： Marked functions or variables can only be used in this contract (Note: the limitation here is only at the code level. Ethereum is a public chain, and anyone can directly obtain the contract status information from the chain)

4．internal: It is generally used in contract inheritance. The parent contract is marked as an internal state variable or function, which can be directly accessed and called by the child contract (it cannot be directly obtained and called externally)

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.7. Denial of service detection【security】

Audit description：Denial of service attack is a DoS attack on Ethereum contract, which makes ether or gas consume a lot. In more serious cases, it can make the contract code logic unable to operate normally. The common causes of DoS attack are: unreasonable design of require check condition, uncontrollable number of for cycles, defects in business logic design, etc.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.8. Conditional competition detection【security】

Audit description： The Ethereum node gathers transactions and forms them into blocks. Once the miners solve the consensus problem, these transactions are considered effective. The miners who solve the block will also choose which transactions from the mine pool will be included in the block. This is usually determined by gasprice transactions. Attackers can observe whether there are transactions in the transaction pool that may contain problem solutions, After that, the attacker can obtain data from this transaction, create a higher-level transaction gasprice, and include its transaction in a block before the original, so as to seize the original solution.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.9. Consistency detection【security】

Audit description： The update logic in smart contract (such as token quantity update, authorized transfer quota update, etc.) is often accompanied by the check logic of the operation object (such as anti overflow check, authorized transfer quota check, etc.), and when the update object is inconsistent with the check object, the check operation may be invalid, Thus, the conditional check logic is ignored and unexpected logic is executed. For example, the authorized transfer function function function transfer from (address _from, address _to, uint256 _value) returns (bool success) is used to authorize others to transfer on behalf of others. During transfer, the permission [_from] [MSG. Sender] authorized transfer limit will be checked, After passing the check, the authorized transfer limit will be updated at the same time of transfer. When the update object in the update logic is inconsistent with the check object in the check logic, the authorized transfer limit of the authorized transfer user will not change, resulting in that the authorized transfer user can transfer all the assets of the authorized account.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.10. Variable coverage detection【security】

Audit description： Smart contracts allow inheritance relationships, in which the child contract inherits all the methods and variables of the parent contract. If a global variable with the same name as the parent contract is defined in the child contract, it may lead to variable coverage and corresponding asset losses.

Audit results： After audit, there is no such security problem.

Safety advice： NONE.

8.2.11. Random number detection【security】

Audit description： Random numbers are often used in smart contracts. When designing the random number generation function, the generation and selection of random seeds should avoid the data information that can be queried on the blockchain, such as block Number and block Timestamp et al. These data are vulnerable to the influence of miners, resulting in the predictability of random numbers to a certain extent.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.12. Numerical operation detection 【security】

Audit description： Solidity supports addition, subtraction, multiplication, division and other conventional numerical operations, but solidty does not support floating-point types. When multiplication and division operations exist at the same time, the numerical operation order should be adjusted reasonably to reduce the error as much as possible.

Audit results： After audit, there is no such security problem.

Safety advice：NONE.

8.2.13. Call injection detection 【security】

Audit description： In the solid language, you can call a contract or a method of a local contract through the call method. There are roughly two ways to call: < address > Call (method selector, arg1, arg2,...) or < address > Call (bytes). When using call call, we can pass method selectors and parameters by passing parameters, or directly pass in a byte array. Based on this function, it is recommended that strict permission check or hard code the function called by call when using call function call.

Audit results： After audit, there is no such security problem.

Safety advice： **NONE.**

8.3. Business logic

Business logic design is the core of smart contract. When using programming language to develop contract business logic functions, developers should fully consider all aspects of the corresponding business, such as parameter legitimacy check, business permission design, business execution conditions, interaction design between businesses, etc.

8.3.1. Initialize Initialize business logic【security】

Audit description： Conduct security audit on the design of initialize business logic in the contract to check whether the function can be called multiple times and whether the caller's identity is checked for permission.

Audit results： The initialize function in the contract is only allowed to be called by factory, which can be used for permission check.

Code file： PancakePair.sol 363~367

Code information：

```
// called once by the factory at time of deployment
    function initialize(address _token0, address _token1) external { //It is used to initiate the contract when creating a transaction pair in the factory contract
        require(msg.sender == factory, 'Pancake: FORBIDDEN'); // Permission check
        token0 = _token0;
        token1 = _token1;
        }
```

Safety advice： **NONE.**

8.3.2. GetReserves business logic design【security】

Audit description： Conduct security audit on getReserves asset information acquisition function in the contract to check whether there are inconsistent security issues and whether there are defects in business logic design.

Audit results： The business logic design of getReserves asset information acquisition in the contract is correct.

Code file： PancakePair.sol 335~339

Code information：

```
    function getReserves() public view returns (uint112 _reserve0, uint112 _reserve1, uint32 _blockTimestampLast) { //Obtain the asset information in the current transaction pair and the latest transaction block time
            _reserve0 = reserve0;
            _reserve1 = reserve1;
            _blockTimestampLast = blockTimestampLast; //Time of the latest transaction block

    }
```

Safety advice：NONE.

8.3.3. _ SafeTransfer transfer business logic【security】

Audit results： For the contract_ SafeTransfer sends token functions for security audit to check whether there are design defects.

Audit results： In the contract_ SafeTransfer token sending function is designed correctly.

Code file： PancakePair.sol 341~344

Code information：

```
    function _safeTransfer(address token, address to, uint value) private {
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR, to,
```

```
value)); //Call through the call. The selector is SELECTOR and transfer
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'Pancake: TRANSFER_FAILED'); //Check whether the call is successful

    }
```

Safety advice：NONE.

8.3.4. _ Update Initialize Business Logic【security】

Audit description： For the contract_ Update Update the business logic design for security audit to check whether there are design defects.

Audit results： Contractual_ Update The business logic design is correct.

Code file： PancakePair.sol 370~383

Code information：

```
    // update reserves and, on the first call per block, price accumulators
    function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1) private { //Update reserves value
        require(balance0 <= uint112(-1) && balance1 <= uint112(-1), 'Pancake: OVERFLOW'); //Check whether the balance value is greater than unit112
        uint32 blockTimestamp = uint32(block.timestamp % 2**32); //Get chunk timestamp
        uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
        if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) { //Check whether timeElapsed is greater than zero_ Reserve0 and_ Whether reserve0 is not zero
            // * never overflows, and + overflow is desired
            price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) * timeElapsed;
            price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) * timeElapsed;
        }
        reserve0 = uint112(balance0);//Update the value of reserve0 in the constant product
        reserve1 = uint112(balance1);//Update the value of reserve1 in the constant product
        blockTimestampLast = blockTimestamp; //Update the block time to the current block time
        emit Sync(reserve0, reserve1); //Trigger synchronization events through emit

    }
```

Safety advice： NONE.

8.3.5. _ MintFee service fee business logic 【security】

Audit description: For the contract_ MintFee service charge calculation business logic is designed for security audit to check whether there are design defects.

Audit results： Contractual_ MintFee service charge calculation business logic design is correct.

Code file: PancakePair.sol 386~404

Code information:

```
    // if fee is on, mint liquidity equivalent to 8/25 of the growth in sqrt(k)
    function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn)
{ //Calculation of service charge
        address feeTo = IPancakeFactory(factory).feeTo(); //Obtain the service charge acceptance
address
        feeOn = feeTo != address(0); //Address non null check
        uint _kLast = kLast; //Record the value of constant product at a certain time in the past
        if (feeOn) { //Judge whether the service charge switch is on
            if (_kLast != 0) {//Judge current_ Whether klast is 0
                uint rootK = Math.sqrt(uint(_reserve0).mul(_reserve1));
                uint rootKLast = Math.sqrt(_kLast);
                if (rootK > rootKLast) {
                    uint numerator = totalSupply.mul(rootK.sub(rootKLast)).mul(8); //molecule
                    uint denominator = rootK.mul(17).add(rootKLast.mul(8)); //denominator
                    uint liquidity = numerator / denominator; //Calculate liquidity
                    if (liquidity > 0) _mint(feeTo, liquidity);
                }
            }
        } else if (_kLast != 0) { //If k is not 0, set it to 0
            kLast = 0;
        }}
```

Safety advice： NONE.

8.3.6. Design of mint service logic【security】

Audit description: Conduct security audit on the mint token issuing function in the contract to check whether there are design defects.

Audit results: The business logic design of mint token issuing function in the contract is correct.

Code file: PancakePair.sol 407~428

Code information:

```
// this low-level function should be called from a contract which performs important safety checks
function mint(address to) external lock returns (uint liquidity) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // Get the current transaction pair's reverse
    uint balance0 = IERC20(token0).balanceOf(address(this)); //Get the balance0
    uint balance1 = IERC20(token1).balanceOf(address(this)); //Get the balance1
    uint amount0 = balance0.sub(_reserve0); //Calculate the number of reverse0 tokens
    uint amount1 = balance1.sub(_reserve1); //Calculate the number of reverse1 tokens
    bool feeOn = _mintFee(_reserve0, _reserve1); //Handling charges
    uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
    if (_totalSupply == 0) { //The total amount of tokens is 0, i.e. the initial provision of liquidity
        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY); //Calculate the fluidity according to the square root of the product in the constant product formula and burn off the initial fluidity
        _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
    } else { //Non initial provision of liquidity
        liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0, amount1.mul(_totalSupply) / _reserve1); //According to the existing liquidity, the amount of liquidity for each token is calculated in proportion, and the minimum value is taken
    }
    require(liquidity > 0, 'Pancake: INSUFFICIENT_LIQUIDITY_MINTED');
    _mint(to, liquidity); //Issue new liquidity to recipients
    _update(balance0, balance1, _reserve0, _reserve1); //Update the value of two assets in the constant product
    if (feeOn) kLast = uint(reserve0).mul(reserve1); // Check whether the service charge is enabled. If it is enabled, update the latest product value
    emit Mint(msg.sender, amount0, amount1); //Trigger events through emit
```

```
    }
```

Safety advice： NONE.

8.3.7. Burn business logic design 【security】

Audit description： Conduct security audit on the burn token destruction function in the contract to check whether there are design defects.

Audit results： The business logic design of the burn token destruction function in the contract is correct.

Code file： PancakePair.sol 431~453

Code information：

```
// this low-level function should be called from a contract which performs important safety checks
    function burn(address to) external lock returns (uint amount0, uint amount1) {
        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // Get the reverse of the current transaction pair
        address _token0 = token0;                                // Get token0 address
        address _token1 = token1;                                // Get token1 address
        uint balance0 = IERC20(_token0).balanceOf(address(this)); //Get the number of token0
        uint balance1 = IERC20(_token1).balanceOf(address(this)); //Get the number of token1
        uint liquidity = balanceOf[address(this)];    //Get the number of liquidity tokens in the current transaction pair contract

        bool feeOn = _mintFee(_reserve0, _reserve1); //Calculation of service charge
        uint _totalSupply = totalSupply; // gas savings, must be defined here since totalSupply can update in _mintFee
        amount0 = liquidity.mul(balance0) / _totalSupply; // using balances ensures pro-rata distribution
        amount1 = liquidity.mul(balance1) / _totalSupply; // using balances ensures pro-rata distribution
        require(amount0 > 0 && amount1 > 0, 'Pancake: INSUFFICIENT_LIQUIDITY_BURNED'); //Asset quantity check
        _burn(address(this), liquidity); //Burn off the liquidity transferred by users in advance
        _safeTransfer(_token0, to, amount0); //Send token0 to the receiver
        _safeTransfer(_token1, to, amount1); //Send token1 to the receiver
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
```

```
        _update(balance0, balance1, _reserve0, _reserve1); //Update the value of two assets with constant product
        if (feeOn) kLast = uint(reserve0).mul(reserve1); // Update Klast
        emit Burn(msg.sender, amount0, amount1, to);

    }
```

Safety advice：NONE.

8.3.8. Swap business logic design【security】

Audit description： Conduct security audit on the business logic design of swap function in the contract to check whether there are design defects.

Audit results： The business logic design of swap function in the contract is correct.

Code file： PancakePair.sol 456~484

Code information：

```
 // this low-level function should be called from a contract which performs important safety checks
    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock { //Asset trading
        require(amount0Out > 0 || amount1Out > 0, 'Pancake: INSUFFICIENT_OUTPUT_AMOUNT'); //Parameter check
        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // Get the reverse quantity of the transaction pair
        require(amount0Out < _reserve0 && amount1Out < _reserve1, 'Pancake: INSUFFICIENT_LIQUIDITY'); //Check whether the quantity to be purchased is less than the reverse quantity available in the current transaction pair
        uint balance0;
        uint balance1;
        { // scope for _token{0,1}, avoids stack too deep errors
        address _token0 = token0;
        address _token1 = token1;
        require(to != _token0 && to != _token1, 'Pancake: INVALID_TO'); //Address parameter check
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // Transfer out to purchase assets
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // Transfer out to purchase assets
```

```
            if (data.length > 0) IPancakeCallee(to).pancakeCall(msg.sender, amount0Out, amount1Out,
data);
            balance0 = IERC20(_token0).balanceOf(address(this)); //Get token0 balance
            balance1 = IERC20(_token1).balanceOf(address(this)); //Get token1 balance
            }
            uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) :
0; //Calculate the amount0 to be transferred in
            uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) :
0; //Calculate the amount1 to be transferred in
            require(amount0In > 0 || amount1In > 0, 'Pancake: INSUFFICIENT_INPUT_AMOUNT');
//Value check
            { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
            uint balance0Adjusted = (balance0.mul(10000).sub(amount0In.mul(25)));    //Deduct 25% of
the transaction fee
            uint balance1Adjusted = (balance1.mul(10000).sub(amount1In.mul(25)));    //Deduct 25% of
the transaction fee
            require(balance0Adjusted.mul(balance1Adjusted)                                          >=
uint(_reserve0).mul(_reserve1).mul(10000**2), 'Pancake: K'); //The product of the new constant
product must be greater than or equal to the old value}
            _update(balance0, balance1, _reserve0, _reserve1); //update
            emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);

      }
```

Safety advice： NONE.

8.3.9. Skim business logic design【security】

Audit description： Conduct security audit on the business logic design of the skim function in the contract to check whether there are design defects.

Audit results： The business logic design of skim function in the contract is correct.

Code file： PancakePair.sol 456~484

Code information：

```
// force balances to match reserves
    function skim(address to) external lock { //When the number of assets in the contract is greater
```

```
than the maximum value of uint112, the user is allowed to propose the difference between the contract asset value and the maximum value of uint112
            address _token0 = token0; // gas savings
            address _token1 = token1; // gas savings
            _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this)).sub(reserve0));
            _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this)).sub(reserve1));
      }
```

Safety advice：NONE.

8.3.10. Sync business logic design【security】

Audit description： Conduct security audit on the business logic design of sync function in the contract to check whether there are design defects.

Audit results： The business logic design of sync function in the contract is correct.

Code file： PancakePair.sol 495~498

Code information：

```
// force reserves to match balances
      function sync() external lock { //Synchronize the number of assets cached in the contract with the current value of the contract. It is mainly used to deal with situations where the ratio is very unreasonable and there is no liquidity provider
            _update(IERC20(token0).balanceOf(address(this)),
IERC20(token1).balanceOf(address(this)), reserve0, reserve1);
      }
```

Safety advice：NONE.

8.3.11. _ mint design service logic【security】

Audit description： For the contract _ mint token issuing function conducts security audit to check whether any coins can be minted and whether there are defects in business logic design.

Audit results： In the contract _ mint token issuing function can only be called internally, and the business logic design is correct.

Code file： PancakePair.sol 145~149

Code information：

```
function _mint(address to, uint value) internal { //Only internal calls are allowed, and additional tokens are issued
        totalSupply = totalSupply.add(value);    //Increase total tokens
        balanceOf[to] = balanceOf[to].add(value);     //Increase the number of assets held by to address
        emit Transfer(address(0), to, value);

    }
```

Safety advice： **NONE.**

8.3.12. _ Burn business logic design【security】

Audit description： For the contract_ burn token destruction function performs a security audit to check whether any number of tokens at any address can be destroyed and whether there is a permission verification defect.

Audit results： In the contract _burn token destruction function only allows internal calls, and the business logic design is correct.

Code file： PancakePair.sol 151~155

Code information：

```
function _burn(address from, uint value) internal {//Only internal calls are allowed to destroy tokens
        balanceOf[from] = balanceOf[from].sub(value);    //Update the number of assets held at the address
        totalSupply = totalSupply.sub(value); //Update total tokens
        emit Transfer(from, address(0), value);

    }
```

Safety advice： **NONE.**

8.3.13. Business logic related to authorization operation【security】

Audit description: Conduct security audit on the design of business logic related to authorized operations in the contract to check whether there are design defects.

Audit results: The design of business logic related to authorization operation in the contract is correct.

Code file: PancakePair.sol 157~160

Code information:

```
function _approve(address owner, address spender, uint value) private {
        allowance[owner][spender] = value;    //Set the authorization limit. Although there is conditional competition risk here, due to the harsh utilization conditions, it is rated as passed
        emit Approval(owner, spender, value);
    }
    function approve(address spender, uint value) external returns (bool) {
        _approve(msg.sender, spender, value); //Authorization operation
        return true;
    }
```

Safety advice: NONE.

8.3.14. Logic design of transfer related business【security】

Audit description: Conduct security audit on the business logic design related to transfer in the contract to check whether there is any defect in the business design.

Audit results: The design of transfer related business logic in the contract is correct.

Code file: PancakePair.sol 162~166

Code information:

```
function _transfer(address from, address to, uint value) private {
        balanceOf[from] = balanceOf[from].sub(value); //Update the number of assets held from
        balanceOf[to] = balanceOf[to].add(value); //Update the number of assets held by to
        emit Transfer(from, to, value);
```

```
    }
    function transferFrom(address from, address to, uint value) external returns (bool) {
        if (allowance[from][msg.sender] != uint(-1)) {
            allowance[from][msg.sender] = allowance[from][msg.sender].sub(value); //Update authorization limit
        }
        _transfer(from, to, value); //Call_ Transfer
        return true;
    }
```

Safety advice: NONE.

8.3.15. Design of authorization verification business logic 【security】

Audit description: Conduct security audit on the design of authorization verification business logic in the contract to check whether there is a risk of being bypassed.

Audit results: The design of authorization verification business logic in the contract is correct.

Code file: PancakePair.sol 186~199

Code information:

```
function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external { //Verification and authorization operations
        require(deadline >= block.timestamp, 'Pancake: EXPIRED'); //Timestamp verification
        bytes32 digest = keccak256(
            abi.encodePacked(
                '\x19\x01',
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadline))
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, 'Pancake: INVALID_SIGNATURE');
        _approve(owner, spender, value);
```

```
}
```

Safety advice： NONE.

9. Appendix:Analysis tools

9.1.Solgraph

Solgraph is used to generate a graph of the call relationship between smart contract functions, which is convenient for quickly understanding the call relationship between smart contract functions.

Project address： https://github.com/raineorshine/solgraph

9.2.Sol2uml

Sol2uml is used to generate the calling relationship between smart contract functions in the form of UML diagram.

Project address： https://github.com/naddison36/sol2uml

9.3.Remix-ide

Remix is a browser based compiler and IDE that allows users to build contracts and debug transactions using the solid language.

Project address： http://remix.ethereum.org

9.4.Ethersplay

Etherplay is a plug-in for binary ninja. It can be used to analyze EVM bytecode and graphically present the function call process.

Project address： https://github.com/crytic/ethersplay

9.5.Mythril

Mythril is a security audit tool for EVM bytecode, and supports online contract audit.

Project address：https://github.com/ConsenSys/mythril

9.6.Echidna

Echidna is a security audit tool for EVM bytecode. It uses fuzzy testing technology and supports integrated use with truss.

Project address：https://github.com/crytic/echidna

10. DISCLAIMERS

Chainlion only issues this report on the facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities. For the facts occurring or existing after the issuance, chainlion cannot judge the security status of its smart contract, and is not responsible for it. The security audit analysis and other contents in this report are only based on the documents and materials provided by the information provider to chainlion as of the issuance of this report. Chainlion assumes that the information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed or reflected inconsistent with the actual situation, chainlion shall not be liable for the losses and adverse effects caused thereby. Chainlion only conducted the agreed safety audit on the safety of the project and issued this report. Chainlion is not responsible for the background and other conditions of the project.

CHAINLION

Blockchain world patron saint building blockchain ecological security