LUNARAY
BLOCKCHAINSECURITY

SMART CONTRACT SECURITY AUDIT REPORT

For W3SWAP

21 January 2023

lunaray.co

Table of Contents

1. Overview

On Jan 17, 2023, the security team of Lunaray Technology received the security audit request of the **W3SWAP project**. The team completed the audit of the **W3SWAP smart contract** on Jan 21, 2023. During the audit process, the security audit experts of Lunaray Technology and the W3SWAP project interface Personnel communicate and maintain symmetry of information, conduct security audits under controllable operational risks, and avoid risks to project generation and operations during the testing process.

Through communicat and feedback with W3SWAP project party, it is confirmed that the loopholes and risks found in the audit process have been repaired or within the acceptable range. The result of this W3SWAP smart contract security audit: **Passed**

Audit Report Hash:
4893D9A4B3C3D9BAB89AF759AB940C51D8C90F198064AE1E369B35AA54843115

2. Background

2.1 Project Description

Project name	W3SWAP
Contract type	Swap
Code language	Solidity
Contract file	MasterChef.sol, PNN.sol, PNNDistributor.sol
Introduction	W3Swap is a decentralized aggregation exchange dedicated to creating a highly liquid aggregated trading ecosystem to provide more valuable liquidity.

2.2 Audit Range

Smart contract file name and corresponding SHA256：

Name	SHA256
PNNDistributor.sol	1C775D7260438969DE09B2F1071E1D295591114E868E29C7DE2C8FC0BF713BC9
PNN.sol	CBAABF5D472D9C10FA507470F05C770D280693FC8AA6F02490D2A592F9662B73
MasterChef.sol	63279FB15B447F402EE5FFBF0567BD6CD53DC220841209DA9737391F1CF2801A

3. Project contract details

3.1 Contract Overview

PNNDistributor Contract

The contract mainly implements the exchange of a certain number of oskDao Token for a PNN Token credential.

PNN Contract

The contract mainly implements the ERC20 standard and its main function is to serve as a credential for adding pools to the MasterChef contract, and the pools can be removed when they expire.

MasterChef Contract

The contract mainly implements the functions of pledging funds, withdrawing funds and withdrawing rewards from the user, as well as a series of configuration items that require advanced privileges to modify, such as the block reward base, the receipt address for fees, adding pools and updating pools.

3.2 Contract details

PNNDistributor Contract

Name	Parameter	Attributes
claimPNN	none	public

PNN Contract

Name	Parameter	Attributes
decimals	none	public
mint	address to uint256 amount	onlyOwner
_transfer	address from address to uint256 amount	internal
updateBlackList	address user bool flag	onlyOwner
updateWhiteList	address user bool flag	onlyOwner

MasterChef Contract

Name	Parameter	Attributes
updateCanAdd	bool _canAdd	onlyOwner
updateMultiplier	uint256 multiplierNumber	onlyOwner
poolLength	none	external
getAllNode	none	public
add	none	public
pendingNodeReward	uint _pid	public
claimNodeReward	uint _pid	public
canWithdrawPNN	uint256 pid address user	public
withdrawPNN	uint256 pid	public
getMultiplier	uint256 _from uint256 _to	public
pendingCake	address _user	external
updatePool	none	public
deposit	uint256 _pid	public
claimReward	none	public
withdraw	uint256 _pid	public

4. Audit details

4.1 Findings Summary

Severity	Found	Resolved	Acknowledged
High	1	0	1
Medium	0	0	0
Low	1	0	1
Info	1	0	1

4.2 Risk distribution

Name	Risk level	Repair status
Administrator Permissions	Low	Acknowledged
Missing event	Info	Acknowledged
Variables are updated	No	normal
Floating Point and Numeric Precision	No	normal
Default visibility	No	normal
tx.origin authentication	No	normal
Faulty constructor	No	normal
Unverified return value	No	normal
Insecure random numbers	No	normal
Timestamp Dependent	No	normal
Transaction order dependency	No	normal
Delegatecall	No	normal
Call	No	normal
Denial of Service	No	normal
Logical Design Flaw	High	Acknowledged
Fake recharge vulnerability	No	normal
Short address attack Vulnerability	No	normal
Uninitialized storage pointer	No	normal
Frozen account bypass	No	normal
Uninitialized	No	normal
Reentry attack	No	normal

Integer Overflow	No	normal

4.3 Risk audit details

4.3.1 Administrator permissions

- **Risk description**

Currently in the contract, only the Owner administrator can set contract-related parameters, which may affect the stability of the project market when the administrator is maliciously manipulated or the private key is leaked.

```
function updateCanAdd(bool _canAdd) public onlyOwner {
    canAdd = _canAdd;
}
function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
    BONUS_MULTIPLIER = multiplierNumber;
}
```

- **Safety advice**

It is recommended to use multi-signature contracts to control administrator privileges, or destroy administrator privileges after the contract is chained.

- **Repair Status**

W3SWAP has Acknowledged.

4.3.2 Missing event

- **Risk description**

Some contract parameter adjustment functions that are only controllable by the administrator and important functions of the contract do not add events, which may make on-chain analysis difficult.

```
function updateCanAdd(bool _canAdd) public onlyOwner {
    canAdd = _canAdd;
}
function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
    BONUS_MULTIPLIER = multiplierNumber;
}
function add() public {
    require(
        pnn.transferFrom(msg.sender, address(this), 1),
```

```
        "can not get PNN"
    );
    require(canAdd, "can not add new Pool");
    poolInfo.push(
        PoolInfo({
            owner: msg.sender,
            lastRewardBlock: block.number,
            isWithdrawn: false,
            endBlock: block.number + nodeRewardBlockCount,
            amount: 0
        })
    );
}
function claimNodeReward(uint _pid) public {
    PoolInfo storage info = poolInfo[_pid];
    require(info.owner == msg.sender, "only owner");
    (uint pending, uint blockNumber) = pendingNodeReward(_pid);
    if (pending > 0) {
        info.lastRewardBlock = blockNumber;
        pg.transfer(msg.sender, pending);
    }
}
function canWithdrawPNN(
    uint256 pid,
    address user
) public view returns (bool) {
    PoolInfo storage p = poolInfo[pid];
    return !p.isWithdrawn && p.owner == user && block.number > p.endBlo
ck;
}
function withdrawPNN(uint256 pid) public {
    require(canWithdrawPNN(pid, msg.sender), "can not withdraw PNN");
    poolInfo[pid].isWithdrawn = true;
    pnn.transfer(msg.sender, 1);
}
```

- **Safety advice**

Add Event.

- **Repair Status**

W3SWAP has Acknowledged.

4.3.3 Variables are updated

- **Risk description**

When there is a contract logic to obtain rewards or transfer funds, the coder mistakenly updates the value of the variable that sends the funds, so that the user can use the value of the variable that is not updated to obtain funds, thus affecting the normal operation of the project.

- **Audit Results : Passed**

4.3.4 Floating Point and Numeric Precision

- **Risk Description**

In Solidity, the floating-point type is not supported, and the fixed-length floating-point type is not fully supported. The result of the division operation will be rounded off, and if there is a decimal number, the part after the decimal point will be discarded and only the integer part will be taken, for example, dividing 5 pass 2 directly will result in 2. If the result of the operation is less than 1 in the token operation, for example, 4.9 tokens will be approximately equal to 4, bringing a certain degree of The tokens are not only the tokens of the same size, but also the tokens of the same size. Due to the economic properties of tokens, the loss of precision is equivalent to the loss of assets, so this is a cumulative problem in tokens that are frequently traded.

- **Audit Results : Passed**

4.3.5 Default Visibility

- **Risk description**

In Solidity, the visibility of contract functions is public pass default. therefore, functions that do not specify any visibility can be called externally pass the user. This can lead to serious vulnerabilities when developers incorrectly ignore visibility specifiers for functions that should be private, or visibility specifiers that can only be called from within the contract itself. One of the first hacks on Parity's multi-signature wallet was the failure to set the visibility of a function, which defaults to public, leading to the theft of a large amount of money.

- **Audit Results : Passed**

4.3.6 tx.origin authentication

- **Risk Description**

tx.origin is a global variable in Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract can make the contract vulnerable to phishing-like attacks.

- **Audit Results : Passed**

4.3.7 Faulty constructor

- **Risk description**

Prior to version 0.4.22 in solidity smart contracts, all contracts and constructors had the same name. When writing a contract, if the constructor name and the contract name are not the same, the contract will add a default constructor and the constructor you set up will be treated as a normal function, resulting in your original contract settings not being executed as expected, which can lead to terrible consequences, especially if the constructor is performing a privileged operation.

- **Audit Results : Passed**

4.3.8 Unverified return value

- **Risk description**

Three methods exist in Solidity for sending tokens to an address: transfer(), send(), call.value(). The difference between them is that the transfer function throws an exception throw when sending fails, rolls back the transaction state, and costs 2300gas; the send function returns false when sending fails and costs 2300gas; the call.value method returns false when sending fails and costs all gas to call, which will lead to the risk of reentrant attacks. If the send or call.value method is used in the contract code to send tokens without checking the return value of the method, if an error occurs, the contract will continue to execute the code later, which will lead to the thought result.

- **Audit Results : Passed**

4.3.9 Insecure random numbers

- **Risk Description**

All transactions on the blockchain are deterministic state transition operations with no uncertainty, which ultimately means that there is no source of entropy or randomness within the blockchain ecosystem. Therefore, there is no random number function like rand() in Solidity. Many developers use future block variables such as block hashes, timestamps, block highs and lows or Gas caps to generate random numbers. These quantities are controlled pass the miners who mine them and are therefore not truly random, so using past or present block variables to generate random numbers could lead to a destructive vulnerability.

- **Audit Results : Passed**

4.3.10 Timestamp Dependency

- **Risk description**

In blockchains, data block timestamps (block.timestamp) are used in a variety of applications, such as functions for random numbers, locking funds for a period of time, and conditional statements for various time-related state changes. Miners have the ability to adjust the timestamp as needed, for example block.timestamp or the alias now can be manipulated pass the miner. This can lead to serious vulnerabilities if the wrong block timestamp is used in a smart contract. This may not be necessary if the contract is not particularly concerned with miner manipulation of block timestamps, but care should be taken when developing the contract.

- **Audit Results : Passed**

4.3.11 Transaction order dependency

- **Risk description**

In a blockchain, the miner chooses which transactions from that pool will be included in the block, which is usually determined pass the gasPrice transaction, and the miner will choose the transaction with the highest transaction fee to pack into the block. Since the information about the transactions in the block is publicly available, an attacker can watch the transaction pool for transactions that may contain problematic solutions, modify or revoke the attacker's privileges or change the state of the contract to the attacker's detriment. The attacker can then take data from this transaction and create a higher-level transaction gasPrice and include its transactions in a block before the original, which will preempt the original transaction solution.

- **Audit Results : Passed**

4.3.12 Delegatecall

- **Risk Description**

In Solidity, the delegatecall function is the standard message call method, but the code in the target address runs in the context of the calling contract, i.e., keeping msg.sender and msg.value unchanged. This feature supports implementation libraries, where developers can create reusable code for future contracts. The code in the library itself can be secure and bug-free, but when run in another application's environment, new vulnerabilities may arise, so using the delegatecall function may lead to unexpected code execution.

- **Audit Results : Passed**

4.3.13 Call

- **Risk Description**

The call function is similar to the delegatecall function in that it is an underlying function provided pass Solidity, a smart contract writing language, to interact with external contracts or libraries, but when the call function method is used to handle an external Standard Message Call to a contract, the code runs in the environment of the external contract/function The call function is used to interact with an external contract or library. The use of such functions requires a determination of the security of the call parameters, and caution is recommended. An attacker could easily borrow the identity of the current contract to perform other malicious operations, leading to serious vulnerabilities.

- **Audit Results : Passed**

4.3.14 Denial of Service

- **Risk Description**

Denial of service attacks have a broad category of causes and are designed to keep the user from making the contract work properly for a period of time or permanently in certain situations, including malicious behavior while acting as the recipient of a transaction, artificially increasing the gas required to compute a function causing gas exhaustion (such as controlling the size of variables in a for loop), misuse of access control to access the private component of the contract, in which the Owners with privileges are modified, progress state based on external calls, use of obfuscation and oversight, etc. can lead to denial of service attacks.

- **Audit Results : Passed**

4.3.15 Logic Design Flaw

- **Risk Description**

In smart contracts, developers design special features for their contracts intended to stabilize the market value of tokens or the life of the project and increase the highlight of the project, however, the more complex the system, the more likely it is to have the possibility of errors. It is in these logic and functions that a minor mistake can lead to serious depasstions from the whole logic and expectations, leaving fatal hidden dangers, such as errors in logic judgment, functional implementation and design and so on.

The possible logic problems in this project are as follows:

```
function deposit(uint256 _pid) public {
    require(_pid != 0, "deposit CAKE by staking");
    require(poolInfo[_pid].owner != address(0), "pool not exsit");
    require(
        userStakeInfo[msg.sender] == 0 || userStakeInfo[msg.sender] ==
_pid,
        "only deposit one pool"
    );
    require(block.number < endBlock, "pool expired");
    UserInfo storage user = userInfo[msg.sender];
    if (userStakeInfo[msg.sender] == 0) {
        userStakeInfo[msg.sender] = _pid;
    }
    updatePool();
    if (user.amount > 0) {
        uint256 pending = user.amount.mul(accCakePerShare).div(1e12).su
b(
            user.rewardDebt
        );
        if (pending > 0) {
            pg.transfer(msg.sender, pending);
        }
    }
    uint256 _amount = pg.balanceOf(msg.sender);
    if (_amount > 0) {
        totalAmount += _amount - user.amount;
        poolInfo[_pid].amount += _amount - user.amount;
        user.amount = _amount;
    }
    user.rewardDebt = user.amount.mul(accCakePerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
```

```
}

function withdraw(uint256 _pid) public {
    require(_pid != 0, "withdraw CAKE by unstaking");
    UserInfo storage user = userInfo[msg.sender];
    uint256 userAmount = user.amount;
    require(userAmount >= 0, "withdraw: not good");
    updatePool();
    uint256 pending = user.amount.mul(accCakePerShare).div(1e12).sub(
        user.rewardDebt
    );
    if (pending > 0) {
        pg.transfer(address(msg.sender), pending);
    }
    user.amount = 0;
    totalAmount -= userAmount;
    poolInfo[_pid].amount -= userAmount;
    userStakeInfo[msg.sender] = 0;
    user.rewardDebt = user.amount.mul(accCakePerShare).div(1e12);
    emit Withdraw(msg.sender, _pid, userAmount);
}
```

The contract uses pg Token for pledge, and does not transfer or lock pg Token when pledging, so if the pg Token contract is a standard ERC20 contract implementation, t here is a user can transfer their own pg Token to other users after pledging can be pl edged again, resulting in the user after pledging deposit can be diverted to pg Token at will, and when the pledge is released, the user still uses the initial pledged amount for reward acquisition.

If the pg Token does not belong to the standard ERC20 and has been customized, yo u need to check in detail whether it interacts with the MasterChef contract, and lock the user to transfer funds and other functions after pledging;

- **Repair Status**

  W3SWAP has confirmed and described that the pg token cannot be transferred, so the risk does not exist.

4.3.16 Fake recharge vulnerability

- **Risk Description**

The success or failure (true or false) status of a token transaction depends on whether an exception is thrown during the execution of the transaction (e.g., using mechanisms such as require/assert/revert/throw). When a user calls the transfer function of a token contract to transfer funds, if the transfer function runs normally without throwing an exception, the transaction will be successful or not, and the status of the transaction will be true. When balances[msg.sender] < _value goes to the else logic and returns false, no exception is thrown, but the transaction acknowledgement is successful, then we believe that a mild if/else judgment is an undisciplined way of coding in sensitive function scenarios like transfer, which will lead to Fake top-up vulnerability in centralized exchanges, centralized wallets, and token contracts.

- **Audit Results : Passed**

4.3.17 Short Address Attack Vulnerability

- **Risk Description**

In Solidity smart contracts, when passing parameters to a smart contract, the parameters are encoded according to the ABI specification. the EVM runs the attacker to send encoded parameters that are shorter than the expected parameter length. For example, when transferring money on an exchange or wallet, you need to send the transfer address address and the transfer amount value. The attacker could send a 19-passte address instead of the standard 20-passte address, in which case the EVM would fill in the 0 at the end of the encoded parameter to make up the expected length, which would result in an overflow of the final transfer amount parameter value, thus changing the original transfer amount.

- **Audit Results : Passed**

4.3.18 Uninitialized storage pointer

- **Risk description**

EVM uses both storage and memory to store variables. Local variables within functions are stored in storage or memory pass default, depending on their type. uninitialized local storage variables could point to other unexpected storage variables in the contract, leading to intentional or unintentional vulnerabilities.

- **Audit Results : Passed**

4.3.19 Frozen Account bypass

- **Risk Description**

In the transfer operation code in the contract, detect the risk that the logical functionality to check the freeze status of the transfer account exists in the contract code and can be passpassed if the transfer account has been frozen.

- **Audit Results : Passed**

4.3.20 Uninitialized

- **Risk description**

The initialize function in the contract can be called pass another attacker before the owner, thus initializing the administrator address.

- **Audit Results : Passed**

4.3.21 Reentry Attack

- **Risk Description**

An attacker constructs a contract containing malicious code at an external address in the Fallback function When the contract sends tokens to this address, it will call the malicious code. The call.value() function in Solidity will consume all the gas he receives when it is used to send tokens, so a re-entry attack will occur when the call to the call.value() function to send tokens occurs before the actual reduction of the sender's account balance. The re-entry vulnerability led to the famous The DAO attack.

- **Audit Results : Passed**

4.3.22 Integer Overflow

- **Risk Description**

Integer overflows are generally classified as overflows and underflows. The types of integer overflows that occur in smart contracts include three types: multiplicative overflows, additive overflows, and subtractive overflows. In Solidity language, variables support integer types in steps of 8, from uint8 to uint256, and int8 to int256, integers specify fixed size data types and are unsigned, for example, a uint8 type , can only be stored in the range 0 to 2^8-1, that is, [0,255] numbers, a uint256 type can only store numbers in the range 0 to 2^256-1. This means that an integer variable can only have a certain range of numbers represented, and cannot exceed this formulated range. Exceeding the range of values expressed pass the variable type will result in an integer overflow vulnerability.

- **Audit Results : Passed**

5. Security Audit Tool

Tool name	Tool Features
Oyente	Can be used to detect common bugs in smart contracts
securify	Common types of smart contracts that can be verified
MAIAN	Multiple smart contract vulnerabilities can be found and classified
Lunaray Toolkit	self-developed toolkit

Disclaimer：

Lunaray Technology only issues a report and assumes corresponding responsibilities for the facts that occurred or existed before the issuance of this report, Since the facts that occurred after the issuance of the report cannot determine the security status of the smart contract, it is not responsible for this.

Lunaray Technology conducts security audits on the security audit items in the project agreement, and is not responsible for the project background and other circumstances, The subsequent on-chain deployment and operation methods of the project party are beyond the scope of this audit.

This report only conducts a security audit based on the information provided by the information provider to Lunaray at the time the report is issued, If the information of this project is concealed or the situation reflected is inconsistent with the actual situation, Lunaray Technology shall not be liable for any losses and adverse effects caused thereby.

There are risks in the market, and investment needs to be cautious. This report only conducts security audits and results announcements on smart contract codes, and does not make investment recommendations and basis.

https://lunaray.co

https://github.com/lunaraySec

https://twitter.com/lunaray_Sec

http://t.me/lunaraySec