P2P & Farm

Smart Contract Audit Report
Prepared for Waggy Finance

Date Issued:	Mar 7, 2022
Project ID:	AUDIT2022003
Version:	v1.0
Confidentiality Level:	Public

inspex

CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2022003
Version	v1.0
Client	Waggy Finance
Project	P2P & Farm
Auditor(s)	Puttimet Thammasaeng Patipon Suwanbol Ronnachai Chaipha
Author(s)	Patipon Suwanbol
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Mar 7, 2022	Full report	Patipon Suwanbol

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 Waggy Finance, Inspex team conducted an audit to verify the security posture of the P2P & Farm smart contracts between Jan 20, 2022 and Jan 24, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of P2P & Farm 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 3 critical, 6 high, 4 medium, 3 low, 2 very low and 2 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that P2P & Farm smart contracts have high-level protections in place to be safe from most attacks.

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

Waggy Finance is a platform or a way for two people to arrange for digital assets with "Proof of Waggrian" which is the third person to audit transactions between buying and selling.

Users can trade crypto currencies with fiat or vice versa directly through Waggy's decentralized peer-to-peer exchange (P2P) with the fraud-preventing mechanism invented by Waggy Finance. The platform also offers yield farming to boost the user's earnings by staking the platform's NFT to earn $WAG.

Scope Information:

Project Name	P2P & Farm
Website	https://waggy.finance
Smart Contract Type	Ethereum Smart Contract
Chain	Harmony One
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jan 20, 2022 - Jan 24, 2022
Reassessment Date	Feb 7, 2022 - Feb 8, 2022 and Mar 1 , 2022 - Mar 2, 2022

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

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

2.2. Scope

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

Initial Audit: (Commit: 60b10e84e9197bdc55f673ad54e969964f678cc2)

Contract	Location (URL)
AvatarNFT	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/farm/AvatarNFT.sol
GasStation	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/farm/GasStation.sol
MasterChef	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/farm/MasterChef.sol
FeeCalculator	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/p2p/FeeCalculator.sol
MerchantMultiToken	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/p2p/MerchantMultiToken.sol
RewardCalculator	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/p2p/RewardCalculator.sol
WNativeRelayer	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/p2p/WNativeRelayer.sol
WaggyToken	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/p2p/WaggyToken.sol
BlackListUser	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/BlackListUser.sol
Validator	https://github.com/inspex-archive/Waggy-p2p/blob/60b10e84e9/contracts/Validator.sol

Reassessment: (Commit: 2f5e5e423a7c49d47e3e04763756dad8557720dc)

Contract	Location (URL)
AvatarNFT	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/farm/AvatarNFT.sol
GasStation	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/farm/GasStation.sol
MasterChef	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/farm/MasterChef.sol

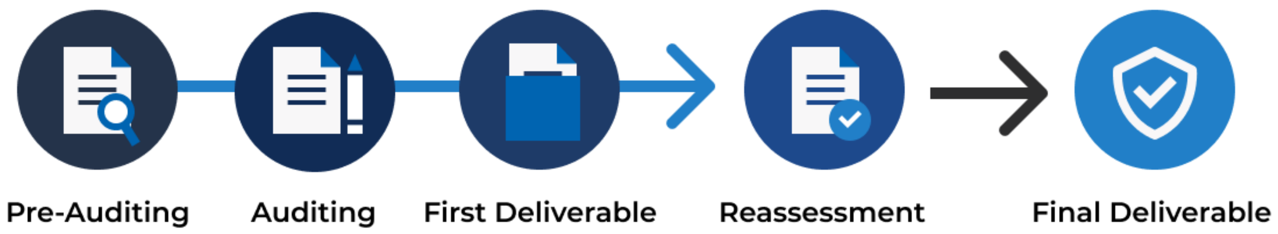
FeeCalculator	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/p2p/FeeCalculator.sol
MerchantMultiToken	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/p2p/MerchantMultiToken.sol
RewardCalculator	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/p2p/RewardCalculator.sol
WNativeRelayer	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/p2p/WNativeRelayer.sol
WaggyToken	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/p2p/WaggyToken.sol
BlackListUser	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/BlackListUser.sol
Validator	https://github.com/WaggyFinance/waggy-p2p/blob/2f5e5e423a/contracts/Validator.sol

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

3. Methodology

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

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



3.1. Test Categories

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

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

3.2. Audit Items

The following audit items were checked during the auditing activity.

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

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

3.3. Risk Rating

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

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

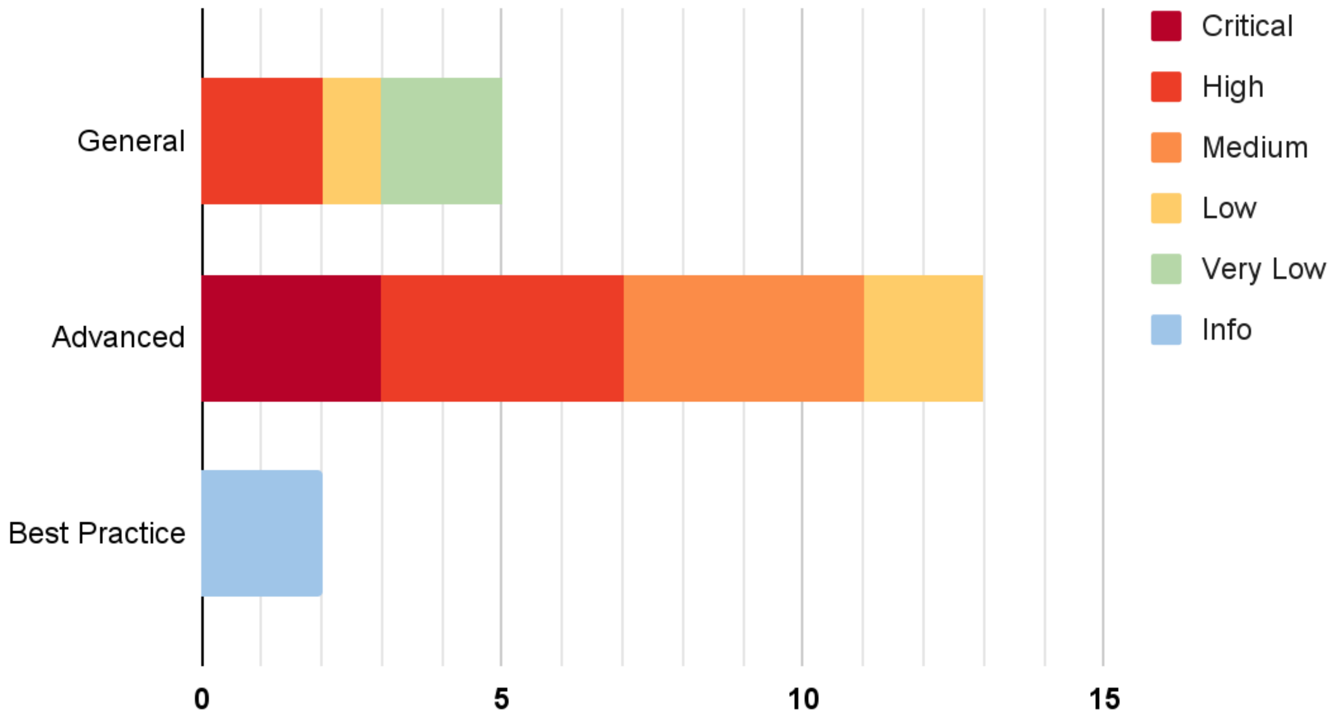
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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

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

4. Summary of Findings

From the assessments, Inspex has found 20 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	Insecure Design in Reward and Fee Calculation	Advanced	Critical	Resolved *
IDX-002	Reward Pool Drained by Manipulating NFT Address	Advanced	Critical	Resolved
IDX-003	Missing user.rewardDebt Update in Claim() Function	Advanced	Critical	Resolved
IDX-004	Missing TokenID Validation	Advanced	High	Resolved
IDX-005	Centralized Control of State Variable	General	High	Resolved *
IDX-006	Use of Upgradable Contract Design	General	High	Resolved *
IDX-007	Improper Updating State Variable to Business Design (user.lastWarning)	Advanced	High	Resolved
IDX-008	Improper Design on Validator Contract	Advanced	High	Resolved *
IDX-009	Improper Reward Calculation (Default Pool Uses Reward Token as lpToken)	Advanced	High	Resolved
IDX-010	Design Flaw in unStake() Function	Advanced	Medium	Resolved
IDX-011	Design Flaw in evaluate() Function	Advanced	Medium	Resolved
IDX-012	Uninitialized Contract State of NFT Price	Advanced	Medium	Resolved
IDX-013	Improper Reward Calculation (_withUpdate Parameter and updateMultiplier() Function)	Advanced	Medium	Resolved
IDX-014	Improper Reward Calculation (Duplicate lpToken)	Advanced	Low	Resolved
IDX-015	Design Flaw in massUpdatePool() Function	General	Low	Resolved
IDX-016	Design Flaw in Reward Distribution Model	Advanced	Low	Resolved
IDX-017	Design Flaw in claimAll() Function	General	Very Low	Resolved
IDX-018	Insufficient Logging for Privileged Functions	General	Very Low	Resolved
IDX-019	Inexplicit Solidity Compiler Version	Best Practice	Info	Resolved
IDX-020	Improper Function Visibility	Best Practice	Info	Resolved

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

5. Detailed Findings Information

5.1. Insecure Design in Reward and Fee Calculation

ID	IDX-001
Target	MerchantMultiToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** Anyone can create a malicious token to make a trade transaction. The malicious token's supply can be designed to be a large amount in order to make the contract minting the huge amount of governance token from the reward calculation formula. **Likelihood: High** It is very likely that this issue will happen since there is no restriction mechanism. It also motivates anyone to perform the attack because the reward is amplified by the amount of token not the value of the token, so the capital required for this commitment is very cheap.
Status	Resolved * Waggy Finance team has mitigated this issue by whitelisting the token and removing the reward for P2P trading. For the fee calculation, since only the whitelisted token is allowed, Waggy Finance team accepts to collect the fee base on the token amount.

5.1.1. Description

The `MerchantMultiToken` contract allows the platform users to buy and sell any token by providing the target token address and the amount to sell.

To begin with, the seller executes the `approveTransaction()` function for locking the tokens and creating a selling transaction respectively.

MerchantMultiToken.sol

```
/*
  Step 2
  When buyer is request to buy the token is had action to approve at seller
  The seller to call function approveTransaction for lock balance, it ready for wait to fait transfer.
  _amount is value of buyer want it.
  */
function approveTransaction(
    ERC20Upgradeable _token,
```

```
    uint256 _amount,
    address _buyer
) public {
    require(getShopBalance(msg.sender, address(_token)) >= _amount, "Balance not enougth");
    // sub avalible shop balance
    setShopBalance(address(_token), msg.sender, getShopBalance(msg.sender, address(_token)).sub(_amount));

    UserInfo storage buyerInfoData = buyerInfo[msg.sender][_buyer];
    uint256 transactionLength = buyerInfoData.transactions.length;
    Transaction storage transaction;
    // check last transaction is finish
    if (transactionLength != 0) {
        transaction = buyerInfoData.transactions[transactionLength.sub(1)];
        require(
            transaction.status == TransactionStatus.FINISH || transaction.status == TransactionStatus.CANCELED,
            "Transaction status mismatch"
        );
    }
    // create new transaction pending add push in transaction list
    buyerInfoData.transactions.push(
        Transaction(_token, TransactionStatus.PENDING_TRANSFER_FAIT, _amount, "", _amount, block.number, block.number, "")
    );
    // update total lock balance
    setTotalLockBalance(msg.sender, address(_token), getTotalLockBalance(msg.sender, address(_token)).add(_amount));
    emit ApproveTransaction(msg.sender, address(_token), _amount);
}
```

Afterwards, the seller will execute the `releaseTokenBySeller()` function to release the locked token and transfer to the buyer wallet address. The buyer's wallet can belong to anyone, for example, another seller's wallet.

MerchantMultiToken.sol

```
/*
Step 3
    For seller release token to buyer when the seller approve a evidence of faite transfer slip
    _address is a receipt waller address
    _amount is value of token to transfer
*/
function releaseTokenBySeller(address _buyer, ERC20Upgradeable _token) public {
    UserInfo storage buyerInfoData = buyerInfo[msg.sender][_buyer];
```

```
    uint256 transactionLength = buyerInfoData.transactions.length;
    require(transactionLength != 0, "Not found transaction");
    Transaction storage transaction =
buyerInfoData.transactions[transactionLength.sub(1)];
    require(transaction.status == TransactionStatus.PENDING_TRANSFER_FAIT,
"Transaction missmatch");
    transaction.lockAmount = 0;
    transaction.status = TransactionStatus.FINISH;
    transaction.updateAt = block.number;

    setTotalLockBalance(
        msg.sender,
        address(transaction.token),
        getTotalLockBalance(msg.sender,
address(transaction.token)).sub(transaction.amount)
    );

    uint256 fee = feeCalculator.calculateFee(transaction.amount);
    uint256 receiverAmount = transaction.amount.sub(fee);
    _token.safeTransfer(feeCollector, fee);
    if (address(_token) == address(wbnb)) {
        _token.safeTransfer(address(wnativeRelayer), receiverAmount);
        wnativeRelayer.withdraw(receiverAmount);
        (bool success, ) = _buyer.call{ value: receiverAmount }("");
        require(success, "WNativeRelayer::onlyWhitelistedCaller:: can't
withdraw");
    } else {
        _token.safeTransfer(_buyer, receiverAmount);
    }

    SuccessTransactionInfo storage successTransactionInfo =
successTransactionCount[msg.sender];
    successTransactionInfo.totalSellAmount =
successTransactionInfo.totalSellAmount.add(transaction.amount);
    successTransactionInfo.totalSellCount =
successTransactionInfo.totalSellCount.add(1);

    // pay reward after complete transaction
    uint256 reward = transaction.amount.mul(700).div(10000);
    gov.mint(msg.sender, reward);
    reward = transaction.amount.mul(300).div(10000);
    gov.mint(_buyer, reward);

    emit ReleaseToken(msg.sender, _buyer, address(_token), transaction.amount,
reward);
}
```

During the `releaseTokenBySeller()` function is executing, the government token reward and transaction fee will be calculated. The token amount of the transaction will be the main factor to calculate the reward and fee amount. Therefore, the attacker can gain more reward amount by using a lot of valueless tokens.

Similarly, in the case of token selling, there will be a fee to charge that transaction to the platform.

However, the fee amount relies on the token amount. This means if the token is not applied the token's decimal standard, which is 10^18, the fee amount will be resulted in miscalculation.

FeeCalculator.sol

```
function calculateFee(uint256 _amount) external pure returns (uint256) {
    if (_amount < 1010000000000000000000) {
        return 0;
    } else if (_amount < 1001000000000000000000) {
        return _amount.mul(25).div(10000);
    } else if (_amount < 10001000000000000000000) {
        return _amount.mul(50).div(10000);
    } else if (_amount < 50001000000000000000000) {
        return _amount.mul(100).div(10000);
    } else {
        return _amount.mul(150).div(10000);
    }
}
```

5.1.2. Remediation

Inspex suggests validating the token address by allowing only whitelisted address and implement a mechanism to check the price of the token, such as a price oracle to ensure the correctness of reward and fee calculation, for example:

- Chainlink (https://docs.chain.link/)
- Uniswap oracles (https://docs.uniswap.org/protocol/V2/concepts/core-concepts/oracles)

5.2. Reward Pool Drained by Manipulating NFT Address

ID	IDX-002
Target	GasStation
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity: Critical** **Impact: High** Users can stake any NFT to the platform. Since the platform applies the `weights` state of the malicious NFT to calculate the reward, the reward can be drained by amplifying the `weights`. In addition, it is also vulnerable to the Reentrancy Attack, causing the reward drained from the pool. **Likelihood: High** It is very likely that the attacker will perform this attack since there is no mechanism to prevent it and the cost of manipulation is very low comparing to the reward gained.
Status	**Resolved** Waggy Finance team has resolved this issue as suggested by whitelisting the NFT address in commit `ed886a72d13c4703e134a27ba8dba3c9c7ba8b05`.

5.2.1. Description

In the `GasStation` contract, the `_nftAddress` parameter is used in the `stake()` and `unstake()` functions without any validation. This allows the attacker to drain the reward in the `GasStation` contract.

In the `stake()` function, the `wnft` state is set from the `_nftAddress` parameter in line 115. Then, the `weight` state is set from the result of `wnft.getWeight()` in line 116 as shown below:

GasStation.sol

```
104 function stake(address _nftAddress, uint256 _tokenId) external {
105     PoolInfo storage pool = poolInfo;
106     UserInfo storage user = userInfo[msg.sender];
107     // claim reward before new staking
108     if (user.weights > 0) {
109         uint256 pending =
    user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
110         if (pending > 0) {
111             pool.lpToken.transfer(address(msg.sender), pending);
112         }
113     }
114 
115     WNFT wnft = WNFT(_nftAddress);
```

```
    uint256 weight = wnft.getWeight();
    require(weight > 0, "can't stake");
    wnft.safeTransferFrom(msg.sender, address(this), _tokenId);
```

After that, the `weight` state is used to calculate the `pool.supply` in line 121 that is shown in the following source code:

GasStation.sol

```
120    if (weight > 0) {
121        pool.supply = pool.supply.add(weight);
122        user.nftStake[_nftAddress] = user.nftStake[_nftAddress].add(1);
123        user.weights = user.weights.add(weight);
124    }
125    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);
126
127    emit Stake(msg.sender, _nftAddress, _tokenId, weight);
128 }
```

Thus, the attacker can deploy their `MaliciousNFT` contract that has the `getWeight()` function to return the huge amount of weight as shown below:

MaliciousNFT.sol

```
function getWeight() external returns (uint256){
    return 1000000;
}
```

After that, the attacker can call the `unStake()` function using the `MaliciousNFT` contract address as `_nftAddress` parameter. The `user.weights` state that is manipulated in the previous step is used for reward calculation, so the attacker will get the huge amount of reward due to the amplified `weights` NFT's state.

GasStation.sol

```
function unStake(address _nftAddress, uint256 _tokenId) external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(user.nftStake[_nftAddress] > 0, "No NFT Stake");
    // Claim reward before unstake
    uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        pool.lpToken.transfer(address(msg.sender), pending);
    }
```

The missing `_nftAddress` validation not only allows the attacker to drain the reward from the `GasStation` contract via manipulating the weight state, but also allows the attacker to drain the reward from this contract by performing Reentrancy Attack.

For Reentrancy Attack, the attacker can deploy the malicious contract that has the `getWeight()` function to repeatedly call the `unStake()` function until the `GasStation` contract is run out of reward as the reward is transferred before updating the state.

5.2.2. Remediation

Inspex suggests validating the NFT address by using whitelisted contract address, for example:

GasStation.sol

```
// NFT Whitelisted address
mapping(address => bool) public isWhitelisted;

function setWhitelistedNFT(address _nftAddress, bool status) public onlyAdmin {
    require(isWhitelisted[_nftAddress] != status, "The whitelisted address is already set.");
    isWhitelisted = status;
}
```

Please note that, the `setWhitelistedNFT()` function is a privileged function that should implement the timelock mechanism for `onlyAdmin` role to delay the change of contract state.

GasStation.sol

```
function stake(address _nftAddress, uint256 _tokenId) external {
    require(isWhitelisted[_nftAddress], "_nftAddress isn't whitelisted.");
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    // claim reward before new staking
    if (user.weights > 0) {
        uint256 pending = user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
        if (pending > 0) {
            pool.lpToken.transfer(address(msg.sender), pending);
        }
    }

    WNFT wnft = WNFT(_nftAddress);
    uint256 weight = wnft.getWeight();
    require(weight > 0, "can't stake");
    wnft.safeTransferFrom(msg.sender, address(this), _tokenId);

    if (weight > 0) {
        pool.supply = pool.supply.add(weight);
        user.nftStake[_nftAddress] = user.nftStake[_nftAddress].add(1);
        user.weights = user.weights.add(weight);
    }
    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

```

```
    emit Stake(msg.sender, _nftAddress, _tokenId, weight);
}

function unStake(address _nftAddress, uint256 _tokenId) external {
    require(isWhitelisted[_nftAddress], "_nftAddress isn't whitelisted.");
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(user.nftStake[_nftAddress] > 0, "No NFT Stake");
    // Claim reward before unstake
    uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        pool.lpToken.transfer(address(msg.sender), pending);
    }

    WNFT wnft = WNFT(_nftAddress);
    uint256 weight = wnft.getWeight();
    user.nftStake[_nftAddress] = user.nftStake[_nftAddress].sub(1);
    user.weights = user.weights.sub(weight);

    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);
    wnft.safeTransferFrom(address(this), msg.sender, _tokenId);
    emit UnStake(msg.sender, _nftAddress, _tokenId, weight);
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.3. Missing rewardDebt Update in Claim() Function

ID	IDX-003
Target	GasStation
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity: Critical** **Impact: High** The `GasStation` contract is not update the claimed reward. Thus, the attacker can call the `claim()` function repeatedly to drain an entire reward in this contract. **Likelihood: High** The `claim()` function can be executed by anyone, so there is no restriction to prevent this issue.
Status	**Resolved** Waggy Finance team has resolved this issue as suggested by updating the `user.rewardDebt` in `claim()` function in commit `ed886a72d13c4703e134a27ba8dba3c9c7ba8b05`.

5.3.1. Description

The `claim()` function is used for claiming the reward of the staked NFT as shown in the following source.

GasStation.sol

```
function claim() external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    // Claim reward before unstake
    uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    require(pending > 0, "No reward");
    pool.lpToken.transfer(address(msg.sender), pending);
}
```

In the source code above, it shows that the `user.rewardDebt` is not updated after the user claims reward. As a result, the attacker can execute the `claim()` function repeatedly to drain the reward from the `GasStation` contract.

5.3.2. Remediation

Inspex suggests updating the `user.rewardDebt` state after the user claims the reward. For example:

GasStation.sol

```
function claim() external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    // Claim reward before unstake
    uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    require(pending > 0, "No reward");
    user.rewardDebt = user.rewardDebt.add(pending);
    pool.lpToken.transfer(address(msg.sender), pending);
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.4. Missing TokenID Validation

ID	IDX-004
Target	GasStation
Category	Advanced Smart Contract Vulnerability
CWE	CWE-20: Improper Input Validation
Risk	**Severity: High** **Impact: High** The staking NFT in the `GasStation` contract can be stolen by other user who has the same NFT contract address. The user, whose NFT got stolen, will not be able to withdraw their NFT back and claim the reward as the transaction will always revert. **Likelihood: Medium** To steal NFT from other users, it is required that the attacker must have `user.weights` state more or equal than the weight of NFT that they want to steal. Hence, the attacker needs to have the accumulated NFT that is worth around the target NFT and in exchange the attacker's NFT will be stuck in the `GasStation` contract.
Status	Resolved Waggy Finance team has resolved this issue as suggested by validating the NFT's ID ownership before `unStake()` function in commit `dfa7f8525ebc2741831119be5bdd93800c980461`.

5.4.1. Description

In the `unStake()` function, the `_tokenId` parameter is used for indicating the NFT to be unstaked as shown in the following source code at line 144:

GasStation.sol

```
130 function unStake(address _nftAddress, uint256 _tokenId) external {
131     PoolInfo storage pool = poolInfo;
132     UserInfo storage user = userInfo[msg.sender];
133     require(user.nftStake[_nftAddress] > 0, "No NFT Stake");
134     // Claim reward before unstake
135     uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
136     if (pending > 0) {
137         pool.lpToken.transfer(address(msg.sender), pending);
138     }
139
140     WNFT wnft = WNFT(_nftAddress);
141     uint256 weight = wnft.getWeight();
142     user.nftStake[_nftAddress] = user.nftStake[_nftAddress].sub(1);
```

```
    user.weights = user.weights.sub(weight);
    wnft.safeTransferFrom(address(this), msg.sender, _tokenId);

    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

    emit UnStake(msg.sender, _nftAddress, _tokenId, weight);
}
```

Since the `unStake()` function does not validate whether the wallet address who executes this function is the owner of the NFT's ID (`_tokenId`) or not, allowing the attacker to use any NFT's ID (`_tokenId`) that they want to steal. However, the restriction is that the attacker must have total `user.weights` more or equal to the NFT that they want to steal and their NFT will be stuck in the `GasStation` contract that comes from staking multiple NFTs.

5.4.2. Remediation

Inspex suggests validating the NFT's ID ownership before allowing the user to manipulate to that NFT, for example:

First, we recommended adding the staked NFT information in the `UserInfo`:

GasStation.sol

```
struct UserInfo {
    mapping(address => uint256) nftStake;
    mapping(address => mapping(uint256 => bool)) stakedNFT; // user.stakedNFT[_nftAddress][_tokenId] = true;
    uint256 weights;
    uint256 rewardDebt; // Reward debt. See explanation below.
}
```

Then, the `stake()` function, the staking NFT information should be added to the `userInfo` state, this includes the token's ownership.

GasStation.sol

```
function stake(address _nftAddress, uint256 _tokenId) external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    // claim reward before new staking
    if (user.weights > 0) {
        uint256 pending = user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
        if (pending > 0) {
            pool.lpToken.transfer(address(msg.sender), pending);
        }
    }

```

```
    WNFT wnft = WNFT(_nftAddress);
    uint256 weight = wnft.getWeight();
    require(weight > 0, "can't stake");
    wnft.safeTransferFrom(msg.sender, address(this), _tokenId);
    user.stakedNFT[_nftAddress][_tokenId] = true;

    if (weight > 0) {
        pool.supply = pool.supply.add(weight);
        user.nftStake[_nftAddress] = user.nftStake[_nftAddress].add(1);
        user.weights = user.weights.add(weight);
    }
    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

    emit Stake(msg.sender, _nftAddress, _tokenId, weight);
}
```

Finally, in the `unStake()` function, it is recommended to add the staked NFT validation and set the `stakedNFT` state.

GasStation.sol

```
function unStake(address _nftAddress, uint256 _tokenId) external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(user.stakedNFT[_nftAddress][_tokenId], "Invalid _tokenId!");
    require(user.nftStake[_nftAddress] > 0, "No NFT Stake");
    // Claim reward before unstake
    uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        pool.lpToken.transfer(address(msg.sender), pending);
    }

    WNFT wnft = WNFT(_nftAddress);
    uint256 weight = wnft.getWeight();
    user.nftStake[_nftAddress] = user.nftStake[_nftAddress].sub(1);
    user.weights = user.weights.sub(weight);
    user.stakedNFT[_nftAddress][_tokenId] = false;
    wnft.safeTransferFrom(address(this), msg.sender, _tokenId);

    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

    emit UnStake(msg.sender, _nftAddress, _tokenId, weight);
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.5. Centralized Control of State Variable

ID	IDX-005
Target	AvatarNFT GasStation MasterChef RewardCalculator WaggyToken WNativeRelayer BlackListUser Validator MerchantMultiToken
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The controlling authorities can change the critical state variables to gain additional profit. For example, the contract owner can call the `ownerClaimToken()` function to transfer all user's funds to the owner's address. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner only.
Status	Resolved * Waggy Finance team has resolved this issue as suggested by adding a timelock contract to delay the contract state change. At the time of the reassessment, the contracts are not yet deployed. The platform users should confirm that only the Timelock has the privileged roles before using the platform.

5.5.1. Description

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

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

The controllable privileged state update functions are as follows:

Target	Contract	Function	Modifier

AvatarNFT.sol (L:34)	AvatarNFT	setPrice()	onlyOwner
GasStation.sol (L:86)	GasStation	setAdmin()	onlyOwner
MasterChef.sol (L:89)	MasterChef	add()	onlyOwner
MasterChef.sol (L:106)	MasterChef	set()	onlyOwner
MasterChef.sol (L:79)	MasterChef	updateMultiplier()	onlyOwner
MasterChef.sol (L:135)	MasterChef	setLockRewardPercent()	onlyOwner
MasterChef.sol (L:135)	MasterChef	dev()	onlyOwner
RewardCalculator.sol (L:27)	RewardCalculator	updateRewardRate()	onlyOwner
WaggyToken.sol (L:73)	WaggyToken	setCap()	onlyGovernor
WaggyToken.sol (L:81)	WaggyToken	setGovernor()	onlyGovernor
WaggyToken.sol (L:88)	WaggyToken	setMinter()	onlyOwner
WaggyToken.sol (L:97)	WaggyToken	revokeRoles()	onlyOwner
WNativeRelayer.sol (L:25)	WNativeRelayer	setCallerOk()	onlyOwner
BlackListUser.sol (L:51)	BlackListUser	revokeRoles()	onlyOwner
BlackListUser.sol (L:57)	BlackListUser	setAdmins()	onlyOwner
Validator.sol (L:111)	Validator	setAdmin()	onlyOwner
Validator.sol (L:115)	Validator	setMinPercent()	onlyOwner
Validator.sol (L:119)	Validator	setMaxPercent()	onlyOwner
MerchantMultiToken.sol (L:137)	MerchantMultiToken	setValidator()	onlyOwner
MerchantMultiToken.sol (L:141)	MerchantMultiToken	setAdmins()	onlyOwner
MerchantMultiToken.sol (L:150)	MerchantMultiToken	setWNativeRelayer()	onlyOwner
MerchantMultiToken.sol (L:154)	MerchantMultiToken	setWBNB()	onlyOwner
MerchantMultiToken.sol (L:158)	MerchantMultiToken	revokeRoles()	onlyOwner
MerchantMultiToken.sol (L:517)	MerchantMultiToken	setBlackList()	onlyOwner
MerchantMultiToken.sol (L:526)	MerchantMultiToken	ownerClaimToken()	onlyOwner
MerchantMultiToken.sol (L:531)	MerchantMultiToken	updateRewardCalculator()	onlyOwner

MerchantMultiToken.sol (L:536)	MerchantMultiToken	updateFeeCalculator()	onlyOwner

5.5.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract, Inspex suggests removing these functions to prevent malicious usage.

However, if modifications are needed, Inspex suggests mitigating this issue by applying the following options:

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

5.6. Use of Upgradable Contract Design

ID	IDX-006
Target	GasStation MerchantMultiToken WaggyToken
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The logic of affected contracts can be arbitrarily changed. This allows the proxy owner to perform malicious actions e.g., stealing the users' funds anytime they want. **Likelihood:** Medium This action can be performed by the proxy owner without any restriction.
Status	Resolved * Waggy Finance team has resolved this issue as suggested by adding a timelock contract to delay the contract upgrade in commit. At the time of the reassessment, the contracts are not yet deployed. The platform users should confirm that only the Timelock has the privileged roles before using the platform.

5.6.1. 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.

GasStation.sol

```
29 contract GasStation is OwnableUpgradeable, ERC721Holder {
30   using SafeMath for uint256;
```

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

The upgradeable contracts are as follows:

Target	Contract
GasStation.sol (L: 29)	GasStation
MerchantMultiToken.sol (L: 57)	MerchantMultiToken
WaggyToken.sol (L: 19)	WaggyToken

5.6.2. Remediation

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

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

5.7. Improper Updating State Variable to Business Design (user.lastWarning)

ID	IDX-007
Target	BlackListUser
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium The `user.lastWarning` is not updated. This means the users who violate the terms of service will not be able to get punished, which is incorrect according to the business design. **Likelihood:** High This issue will happen when the users get warned through the `warningUser()` function. It is unavoidable since the code logic is implemented incorrectly.
Status	Resolved Waggy Finance team has resolved this issue as suggested by updating `user.lastWarning` state in commit `dc0cd62ec75ed998facc2f052c2c34451afde2f8`.

5.7.1. Description

The `BlackListUser` contract can be used to manage a user's status before interacting with the platforms.

When the `warningUser()` function is executed to warn the user, it counts the warning amount before suspending the users if the amount reaches the platform's criteria.

BlackListUser.sol

```
// set warning user count.
function warningUser(address _user) external  {
    require(hasRole(ADMIN_ROLE, msg.sender), "DOES_NOT_HAVE_MINTER_ROLE");
    UserInfo storage user = userInfo[_user];
    require(user.status == STATUS.NORMAL, "Can't warning not normal status
user.");

    uint256 diffTime = block.timestamp.sub(user.lastWarning).div(1 days);
    if (diffTime == 0) {
        user.amount = user.amount.add(1);
        if (user.amount >= ALLOW_LIMIT_TEMPORARY) {
            user.status = STATUS.TEMPORARY;
            user.amount = 0;
            user.totalWarning = user.totalWarning.add(1);
            user.lastWarning = block.timestamp;
```

```
            if (user.totalWarning >= ALLOW_LIMIT_SUSPEND) {
                user.status = STATUS.SUSPEND;
                user.suspendAt = block.timestamp;
            }
        }
    } else {
        user.amount = 1;
    }
}
```

Following the code above, the `user.lastWarning` variable default value is 0 which means in the case that `block.timestamp` is greater than `86400` (1 days).
The result of `block.timestamp.sub(user.lastWarning).div(1 days)` will always greater than 0.

As a result, the `user.amount` will be always updated to 1, causing the users who violate the terms of service will not be able to get punished.

5.7.2. Remediation

Inspex suggests updating the `user.lastWarning` state in the `warningUser()` function, for example:

BlackListUser.sol

```
// set warning user count.
function warningUser(address _user) external  {
    require(hasRole(ADMIN_ROLE, msg.sender), "DOES_NOT_HAVE_MINTER_ROLE");
    UserInfo storage user = userInfo[_user];
    require(user.status == STATUS.NORMAL, "Can't warning not normal status 
user.");

    uint256 diffTime = block.timestamp.sub(user.lastWarning).div(1 days);
    if (diffTime == 0) {
        user.amount = user.amount.add(1);
        if (user.amount >= ALLOW_LIMIT_TEMPORARY) {
            user.status = STATUS.TEMPORARY;
            user.amount = 0;
            user.totalWarning = user.totalWarning.add(1);
            user.lastWarning = block.timestamp;
            if (user.totalWarning >= ALLOW_LIMIT_SUSPEND) {
                user.status = STATUS.SUSPEND;
                user.suspendAt = block.timestamp;
            }
        }
    } else {
        user.amount = 1;
        user.lastWarning = block.timestamp;
    }
}
```

5.8. Improper Design on Validator Contract

ID	IDX-008
Target	Validator
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The answer of all voters is publicly known to anyone, causing the unfair judgment in that case. This is because the majority of answers will be decided as the vote winners and get the reward as the correct vote choice. Hence, the suspect from the appealed case and the users who are in minority vote will lose their money even though they are right. **Likelihood:** Medium It is likely that the voters will check the answers of that case before voting as the winners to get the reward. However, the case will be finalized when the admin confirms the case result, this means by voting following the majority, there is a chance that the voters will lose their collateral from the vote.
Status	Resolved * Waggy Finance team has mitigated this issue by hashing the user answer with secret on off-chain. After that, submitting the previous hashed as the user answer. The wallet address with the `onlyOwner` role will execute the `evaluate()` function with secret to evaluate the answer. Since the secret is controlled by off-chain, the owner of the server can view the secret that is used for each case.

5.8.1. Description

In the `Validator` contract, it allows the users as the evidence-proof troops to judge the appealed case whether the buyer or the seller is cheating. This is called as the "Proof of Waggrian", according to the document.

The evidence-proof troops can judge the pending case by staking their collateral and provide the answer whether who is legit through the `play()` function.

Validator.sol

```
function play(
    string memory _key,
    uint256 _amount,
    bytes32 _answer,
    string memory _remark
```

```
  ) external {
    CaseInfo storage caseInfo = casesInfo[_key];
    require(caseInfo.status == CaseStatus.INPROGRESS, "Can't Vote");
    require(caseInfo.totalValue > caseInfo.currentValue, "The case is closed");
    UserReplyAnswer memory userReplyAnswer =
caseInfo.usersReplyAnswer[msg.sender];
    require(userReplyAnswer.createdAt == 0, "Not allow user reply again");

    uint256 totalValue = caseInfo.totalValue;
    uint256 maxAmount = totalValue.mul(maxPercentValue).div(100);
    uint256 minAmount = totalValue.mul(minPercentValue).div(100);
    // check amount in range
    require(_amount <= maxAmount && _amount >= minAmount, "amount is not in
range limit.");
    // transfer
    ERC20(caseInfo.token).safeTransferFrom(msg.sender, address(this), _amount);
    // add collateral
    totalCollateral = totalCollateral.add(_amount);
    // save reply
    userReplyAnswer.amount = _amount;
    userReplyAnswer.answer = _answer;
    userReplyAnswer.remark = _remark;
    userReplyAnswer.createdAt = block.timestamp;
    caseInfo.usersReplyAnswer[msg.sender] = userReplyAnswer;
    caseInfo.users.push(msg.sender);
    // update progress
    caseInfo.currentValue = caseInfo.currentValue.add(_amount);
    // emit event
    emit UserDecision(msg.sender, _key, _amount, _answer, _remark);

    if (caseInfo.currentValue >= caseInfo.totalValue) {
      emit CaseVoteDone(_key);
    }
}
```

However, the `_answer` from the voters can be known publicly by anyone who monitors the mempool as the format of answer is either `keccak256(abi.encodePacked(BUYER, _key, addressToString(userAddress)))` or `keccak256(abi.encodePacked(SELLER, _key, addressToString(userAddress)))` as in the `evaluate()` function.

Validator.sol

```
function evaluate(string memory _key)
    public
    onlyAdmin
    returns (
      string memory,
```

```
      uint256,
      uint256,
      uint256
    )
  {
    CaseInfo storage caseInfo = casesInfo[_key];
    require(caseInfo.currentValue >= caseInfo.totalValue, "User vote not done.");
    require(caseInfo.users.length > 0, "Case not exist");
    require(caseInfo.resultAt == 0, "This case already had result.");
    uint256 buyyerValueCount;
    uint256 sellerValueCount;
    bytes32 buyerAnswer;
    for (uint256 i = 0; i < caseInfo.users.length; i++) {
      address userAddress = caseInfo.users[i];
      UserReplyAnswer memory userReplyAnswer = caseInfo.usersReplyAnswer[userAddress];
      buyerAnswer = keccak256(abi.encodePacked(BUYER, _key, addressToString(userAddress)));
      if (userReplyAnswer.answer == buyerAnswer) {
        buyyerValueCount = buyyerValueCount.add(userReplyAnswer.amount);
      } else {
        sellerValueCount = sellerValueCount.add(userReplyAnswer.amount);
      }
    }
    if (buyyerValueCount > sellerValueCount) {
      caseInfo.result = BUYER;
    } else if (buyyerValueCount < sellerValueCount) {
      caseInfo.result = SELLER;
    } else {
      caseInfo.result = EQUIVALENT;
    }

    uint256 winnerAmount;
    uint256 fund;
    for (uint256 i = 0; i < caseInfo.users.length; i++) {
      address userAddress = caseInfo.users[i];
      UserReplyAnswer storage userReplyAnswer = caseInfo.usersReplyAnswer[userAddress];
      bytes32 correctAnswer = keccak256(abi.encodePacked(caseInfo.result, _key, addressToString(userAddress)));
      if (userReplyAnswer.answer != correctAnswer) {
        fund = fund.add(userReplyAnswer.amount);
      } else {
        userReplyAnswer.receiveReward = true;
        winnerAmount = winnerAmount.add(1);
      }
```

```
    }
    caseInfo.winnerAmount = winnerAmount;
    caseInfo.status = CaseStatus.SUMMARY;
    caseInfo.fund = fund.sub(fund.mul(10).div(100));
    caseInfo.resultAt = block.timestamp;
    totalCollateral = totalCollateral.sub(caseInfo.currentValue);

    emit EvaluateResult(_key, caseInfo.result, buyyerValueCount, sellerValueCount, caseInfo.resultAt);

    return (caseInfo.result, buyyerValueCount, sellerValueCount, caseInfo.resultAt);
}
```

Hence, the judgment of the cases is not truly transparent because the majority of votes will judge the case result and get the reward as in the business design.

5.8.2. Remediation

Inspex suggests applying the zero-knowledge proof to ensure that the answer (`_answer`) from the voters cannot be known by the other parties, for example using the commit-reveal scheme for more information can be found at https://en.wikipedia.org/wiki/Commitment_scheme.

5.9. Improper Reward Calculation (Default Pool Uses Reward Token as lpToken)

ID	IDX-009
Target	MasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** Medium The reward of the pool that has the same staking token as the reward token will be lower than what it should be. **Likelihood:** High There is a pool that has $WAG as the staking token from the contract constructor. The $WAG is used as the reward token in the `MasterChef` contract, so the reward miscalculation is unavoidable.
Status	Resolved Waggy Finance team has resolved this issue as suggested by adding the `wagPool` address to store the reward in commit `7793a329b3f8836a0be0f03f5ce4c95e7f5d9aa7`.

5.9.1. Description

In the `MasterChef` contract, when the contract is created, the first pool (`poolInfo[0]`) is added automatically as in the contract constructor. This pool uses $WAG token (reward token) as the `lpToken` for allowing the users to stake.

MasterChef.sol

```
constructor(
    address _wag,
    address _devaddr,
    uint256 _wagPerBlock,
    uint256 _startBlock
  ) {
    BONUS_MULTIPLIER = 1;
    lockRewardPercent = 900; //90%
    wag = WaggyToken(_wag);
    devaddr = _devaddr;
    wagPerBlock = _wagPerBlock;
    startBlock = _startBlock;

    // staking pool
```

```
    poolInfo.push(PoolInfo({ lpToken: ERC20(_wag), allocPoint: 1000,
lastRewardBlock: startBlock, accWagPerShare: 0 }));
    totalAllocPoint = 1000;
}
```

As described in `IDX-014 Improper Reward Calculation (Duplicate lpToken)`, the reward calculation applies the balance of `lpToken` of that pool, in this case $WAG token.

Since the $WAG will be minted every time when the reward calculation is executed as in the `updatePool()` function.

MasterChef.sol

```
// Update reward variables of the given pool to be up-to-date.
function updatePool(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    if (block.number <= pool.lastRewardBlock) {
        return;
    }
    uint256 lpSupply = pool.lpToken.balanceOf(address(this));
    if (lpSupply == 0) {
        pool.lastRewardBlock = block.number;
        return;
    }
    uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
    uint256 wagReward =
multiplier.mul(wagPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
    wag.mint(devaddr, wagReward.div(10));
    wag.mint(address(this), wagReward);
    pool.accWagPerShare =
pool.accWagPerShare.add(wagReward.mul(1e12).div(lpSupply));
    pool.lastRewardBlock = block.number;
}
```

Hence, the reward for the first pool (`poolInfo[0]`) will absolutely be inflated.

5.9.2. Remediation

Inspex suggests minting the reward token to another contract to prevent the amount of the staked token from being mixed up with the reward token, or implementing the wrapped token of $WAG as the deposit token (`lpToken`).

5.10. Design Flaw in unStake() Function

ID	IDX-010
Target	GasStation
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The user's shares (NFT) for the affected pools will be getting less continuously since the `pool.supply` is not deducted when the user withdraws their shares (NFT). **Likelihood:** Medium This issue will occur when the users decide to stop staking their NFTs through the `unStake()` function.
Status	Resolved Waggy Finance team has resolved this issue as suggested by updating the remaining shares in commit `0f4ec54365b4c2af3506a86d0422647d657f4929`.

5.10.1. Description

The `GasStation` contract allows the users to stake NFTs and receive reward tokens. When the users stake their NFTs, the `pool.supply` will be increased which represents all active shares in the staking pools so that which affects the reward calculation for each user as in line 121.

GasStation.sol

```
104 function stake(address _nftAddress, uint256 _tokenId) external {
105     PoolInfo storage pool = poolInfo;
106     UserInfo storage user = userInfo[msg.sender];
107     // claim reward before new staking
108     if (user.weights > 0) {
109         uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
110         if (pending > 0) {
111             pool.lpToken.transfer(address(msg.sender), pending);
112         }
113     }
114
115     WNFT wnft = WNFT(_nftAddress);
116     uint256 weight = wnft.getWeight();
117     require(weight > 0, "can't stake");
118     wnft.safeTransferFrom(msg.sender, address(this), _tokenId);
119
```

```
    if (weight > 0) {
        pool.supply = pool.supply.add(weight);
        user.nftStake[_nftAddress] = user.nftStake[_nftAddress].add(1);
        user.weights = user.weights.add(weight);
    }
    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

    emit Stake(msg.sender, _nftAddress, _tokenId, weight);
}
```

The users can also withdraw their NFTs through the `unStake()` function.

GasStation.sol

```
function unStake(address _nftAddress, uint256 _tokenId) external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(user.nftStake[_nftAddress] > 0, "No NFT Stake");
    // Claim reward before unstake
    uint256 pending = user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
      pool.lpToken.transfer(address(msg.sender), pending);
    }

    WNFT wnft = WNFT(_nftAddress);
    uint256 weight = wnft.getWeight();
    user.nftStake[_nftAddress] = user.nftStake[_nftAddress].sub(1);
    user.weights = user.weights.sub(weight);
    wnft.safeTransferFrom(address(this), msg.sender, _tokenId);

    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

    emit UnStake(msg.sender, _nftAddress, _tokenId, weight);
}
```

However, the `pool.supply` is not deducted by the withdrawn weight, causing the removing shares remain in the contract, which affects the reward calculation, in this case `pool.accWagPerShare`, as shown in the `refillPool()` function.

GasStation.sol

```
// Refill reward in pool
function refillPool(uint256 _amount) public {
    PoolInfo storage pool = poolInfo;
    pool.lpToken.transferFrom(msg.sender, address(this), _amount);
    pool.accWagPerShare = pool.accWagPerShare.add(_amount.mul(1e12).div(pool.supply));
```

```
    pool.lastRewardBlock = block.number;
}
```

5.10.2. Remediation

Inspex suggests updating the remaining shares in the withdrawn pool in the `unStake()` function, for example:

GasStation.sol

```
function unStake(address _nftAddress, uint256 _tokenId) external {
    PoolInfo storage pool = poolInfo;
    UserInfo storage user = userInfo[msg.sender];
    require(user.nftStake[_nftAddress] > 0, "No NFT Stake");
    // Claim reward before unstake
    uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        pool.lpToken.transfer(address(msg.sender), pending);
    }

    WNFT wnft = WNFT(_nftAddress);
    uint256 weight = wnft.getWeight();
    pool.supply = pool.supply.sub(weight);
    user.nftStake[_nftAddress] = user.nftStake[_nftAddress].sub(1);
    user.weights = user.weights.sub(weight);
    wnft.safeTransferFrom(address(this), msg.sender, _tokenId);

    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

    emit UnStake(msg.sender, _nftAddress, _tokenId, weight);
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.11. Design Flaw in evaluate() Function

ID	IDX-011
Target	Validator
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** High If the case result from judgment by the users is "EQUIVALENT", the users who vote in that case will not be able to claim their collateral back. Furthermore, the "EQUIVALENT" case result's reward can be exploited to be claimed by a specific setup. **Likelihood:** Low The "EQUIVALENT" case result will happen when the "BUYER" vote has the same amount as the "SELLER" vote. For the specific setup for claiming the reward in "EQUIVALENT" case, it is the amount of "BUYER" vote in that case must have 1 vote higher than "SELLER" vote, the attacker must be the last on who votes "EQUIVALENT", and admin must approve that case.
Status	Resolved Waggy Finance team has resolved this issue as suggested by refunding when the `caseInfo.result` is `EQUIVALENT` in commit `19cefe7203fde2ca7fc41fa94dc5e6305796eb12` .

5.11.1. Description

In the `Validator` contract, it allows the users, the evidence-proof troops, to judge the appealed case whether the buyer or the seller is cheating. This is called as the "Proof of Waggrian", according to the document (https://docs.waggy.finance/products-and-features/proof-of-waggrian).

To judge the appealed case, they must stake their collateral and provide the answer whether who is legit through the `play()` function, which will receive the reward and collateral back when the answer is correct.

Validator.sol

```
function play(
    string memory _key,
    uint256 _amount,
    bytes32 _answer,
    string memory _remark
  ) external {
    CaseInfo storage caseInfo = casesInfo[_key];
    require(caseInfo.status == CaseStatus.INPROGRESS, "Can't Vote");
    require(caseInfo.totalValue > caseInfo.currentValue, "The case is closed");
```

```
    UserReplyAnswer memory userReplyAnswer = caseInfo.usersReplyAnswer[msg.sender];
    require(userReplyAnswer.createdAt == 0, "Not allow user reply again");

    uint256 totalValue = caseInfo.totalValue;
    uint256 maxAmount = totalValue.mul(maxPercentValue).div(100);
    uint256 minAmount = totalValue.mul(minPercentValue).div(100);
    // check amount in range
    require(_amount <= maxAmount && _amount >= minAmount, "amount is not in range limit.");
    // transfer
    ERC20(caseInfo.token).safeTransferFrom(msg.sender, address(this), _amount);
    // add collateral
    totalCollateral = totalCollateral.add(_amount);
    // save reply
    userReplyAnswer.amount = _amount;
    userReplyAnswer.answer = _answer;
    userReplyAnswer.remark = _remark;
    userReplyAnswer.createdAt = block.timestamp;
    caseInfo.usersReplyAnswer[msg.sender] = userReplyAnswer;
    caseInfo.users.push(msg.sender);
    // update progress
    caseInfo.currentValue = caseInfo.currentValue.add(_amount);
    // emit event
    emit UserDecision(msg.sender, _key, _amount, _answer, _remark);

    if (caseInfo.currentValue >= caseInfo.totalValue) {
      emit CaseVoteDone(_key);
    }
}
```

After that, the authorized party, `onlyAdmin`, will execute the `evaluate()` function to judge the pending case. The result will be based on the majority of answers that the evidence-proof troops vote for.

Validator.sol

```
// System order to evaluate
function evaluate(string memory _key)
    public
    onlyAdmin
    returns (
        string memory,
        uint256,
        uint256,
        uint256
    )
  {
    CaseInfo storage caseInfo = casesInfo[_key];
```

```
    require(caseInfo.currentValue >= caseInfo.totalValue, "User vote not done.");
    require(caseInfo.users.length > 0, "Case not exist");
    require(caseInfo.resultAt == 0, "This case already had result.");
    uint256 buyyerValueCount;
    uint256 sellerValueCount;
    bytes32 buyerAnswer;
    for (uint256 i = 0; i < caseInfo.users.length; i++) {
        address userAddress = caseInfo.users[i];
        UserReplyAnswer memory userReplyAnswer = caseInfo.usersReplyAnswer[userAddress];
        buyerAnswer = keccak256(abi.encodePacked(BUYER, _key, addressToString(userAddress)));
        if (userReplyAnswer.answer == buyerAnswer) {
            buyyerValueCount = buyyerValueCount.add(userReplyAnswer.amount);
        } else {
            sellerValueCount = sellerValueCount.add(userReplyAnswer.amount);
        }
    }
    if (buyyerValueCount > sellerValueCount) {
        caseInfo.result = BUYER;
    } else if (buyyerValueCount < sellerValueCount) {
        caseInfo.result = SELLER;
    } else {
        caseInfo.result = EQUIVALENT;
    }

    uint256 winnerAmount;
    uint256 fund;
    for (uint256 i = 0; i < caseInfo.users.length; i++) {
        address userAddress = caseInfo.users[i];
        UserReplyAnswer storage userReplyAnswer = caseInfo.usersReplyAnswer[userAddress];
        bytes32 correctAnswer = keccak256(abi.encodePacked(caseInfo.result, _key, addressToString(userAddress)));
        if (userReplyAnswer.answer != correctAnswer) {
            fund = fund.add(userReplyAnswer.amount);
        } else {
            userReplyAnswer.receiveReward = true;
            winnerAmount = winnerAmount.add(1);
        }
    }
    caseInfo.winnerAmount = winnerAmount;
    caseInfo.status = CaseStatus.SUMMARY;
    caseInfo.fund = fund.sub(fund.mul(10).div(100));
    caseInfo.resultAt = block.timestamp;
    totalCollateral = totalCollateral.sub(caseInfo.currentValue);
```

```

    emit EvaluateResult(_key, caseInfo.result, buyyerValueCount,
sellerValueCount, caseInfo.resultAt);

    return (caseInfo.result, buyyerValueCount, sellerValueCount,
caseInfo.resultAt);
}
```

The collateral and reward will be able to be claimed through the `userClaimReward()` function.

Validatior.sol

```
function userClaimReward(string memory _key) public delay15mins(_key) {
    CaseInfo storage caseInfo = casesInfo[_key];
    require(caseInfo.status == CaseStatus.DONE, "Status is wrong");
    UserReplyAnswer storage user = caseInfo.usersReplyAnswer[msg.sender];
    require(user.receiveReward, "you lose.");

    user.receiveReward = false;

    uint256 reward = user.amount;
    if (caseInfo.fund > 0) {
        reward = reward.add(caseInfo.fund.div(caseInfo.winnerAmount));
    }
    // distribute gov reward.
    uint256 govReward = reward.mul(25).div(10000);
    gov.mint(msg.sender, govReward);
    ERC20(caseInfo.token).safeTransfer(msg.sender, reward);

}
```

However, if the case result is "EQUIVALENT", there are 2 points to concerns:

1. Collateral is not able to be claim back for voters

According to the business design, the collateral for the user who votes the incorrect choice will be transformed into the reward for the users who vote the correct choice.

However, when the case result is "EQUIVALENT", the user who votes "BUYER" or "SELLER" will not be able to claim their collateral back.

2. Claiming the reward in the "EQUIVALENT" case

This happens when the case result is "EQUIVALENT", which there is no mechanism to prevent the "EQUIVALENT" answer from the voters. Hence, a user, who is the last voter, can answer "EQUIVALENT" when that case is required 1 more answer in opposite site of "BUYER", causing the case to result in "EQUIVALENT", so that user is eligible to claim the reward through the `userClaimReward()` function.

5.11.2. Remediation

Inspex suggests adding a validating mechanism that will not count the "EQUIVALENT" answer as the legit vote and refund the collateral back to the users who vote in the case that results in "EQUIVALENT".

5.12. Uninitialized Contract State of NFT Price

ID	IDX-012
Target	AvatarNFT
Category	Advanced Smart Contract Vulnerability
CWE	CWE-908: Use of Uninitialized Resource
Risk	**Severity:** Medium **Impact:** Medium The `AvatarNFT` contract does not set the NFT price in the `constructor()` on deployment. When the contract is deployed, anyone can call the `mint()` function before the `setPrice()` function is executed. However, if the contract owner detects the illegal NFTs, the contract owner can simply deploy the new NFT contract and return the money back to the legit platform users. **Likelihood:** Medium The NFT can be minted by anyone who monitors the mempool then executes the `mint()` function after the contract deployment immediately. However, it requires the address of the contract deployer.
Status	Resolved Waggy Finance team has resolved this issue as suggested by setting the NFT price in the constructor in commit `ee0845f6325b65b83e57802ab2dafdee51a4bd3a`.

5.12.1. Description

In the `AvatarNFT` contract, the `nftPrice` state is not set in the `constructor`. Since the `nftPrice` type is `uint256`, its state is set to 0 by default that is shown in the following source code:

AvatarNFT.sol

```
contract AvatarNFT is Ownable, ERC721URIStorage {
    using SafeMath for uint256;
    using Counters for Counters.Counter;
    using Strings for uint256;

    event Mint(address, uint256);

    Counters.Counter private _tokenIds;
    string public baseURI;
    uint256 private nftPrice;
    mapping(address => uint256) public userOwnerTokenId;

    constructor(string memory _name, string memory _symbol) ERC721(_name,
_symbol) {}
```

According to the contract design, the wallet address with `onlyOwner` role must execute the `setPrice()` function to set the price of the NFT after the contract is deployed as shown below:

AvatarNFT.sol

```
function setPrice(uint256 _price) external onlyOwner {
    nftPrice = _price;
}
```

As a result, anyone who monitors the contract deployment, and executes the `mint()` function after the contract is deployed can mint an NFT for free.

5.12.2. Remediation

Inspex suggests adding the `_setPrice()` function in the `constructor()` to set the `nftPrice` state on the contract deployment as shown below:

AvatarNFT.sol

```
constructor(string memory _name, string memory _symbol, uint256 _price)
ERC721(_name, _symbol) {
    _setPrice(_price);
}

function _setPrice(uint256 _price) internal {
    nftPrice = _price;
}

function setPrice(uint256 _price) external onlyOwner {
    _setPrice(_price);
}
```

5.13. Improper Reward Calculation (_withUpdate Parameter and updateMultiplier() Function)

ID	IDX-013
Target	MasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The $WAG reward miscalculation can lead to unfair $WAG token distribution. **Likelihood:** Medium This issue happens whenever the `totalAllocPoint` is modified and the `_withUpdate` parameter is set to false for the `add()` and `set()` functions. This includes the `updateMultiplier()` function when it is executed without updating all pools through `massUpdatePools()` function first.
Status	Resolved Waggy Finance team has resolved this issue as suggested by removing the `_withUpdate` parameter and calling the `massUpdatePools()` and `updateStakingPool()` function before changing the `BONUS_MULTIPLIER` state in commit `f4a8af6c119b8199d018895ef005b42b272c12d2`.

5.13.1. Description

The `totalAllocPoint` variable is used to determine the portion that each pool would get from the total rewards minted, so it is one of the main factors used in the rewards calculation. Therefore, whenever the `totalAllocPoint` variable is modified without updating the pending rewards first, the reward of each pool will be incorrectly calculated.

In the `add()` and the `set()` functions shown below, if the `_withUpdate` is set to false, the `totalAllocPoint` variable will be modified without updating the rewards.

MasterChef.sol

```
// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up
if you do.
function add(
    uint256 _allocPoint,
    ERC20 _lpToken,
    bool _withUpdate
  ) public onlyOwner {
```

```
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({ lpToken: _lpToken, allocPoint: _allocPoint, lastRewardBlock: lastRewardBlock, accWagPerShare: 0 })
    );
    updateStakingPool();
}
```

MasterChef.sol

```
// Update the given pool's CAKE allocation point. Can only be called by the owner.
function set(
    uint256 _pid,
    uint256 _allocPoint,
    bool _withUpdate
  ) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
    poolInfo[_pid].allocPoint = _allocPoint;
    if (prevAllocPoint != _allocPoint) {
        totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
        updateStakingPool();
    }
}
```

For the `updateMultiplier()` function, the contract owner can execute it directly without updating all pending rewards in all pools beforehand.

MasterChef.sol

```
function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
    BONUS_MULTIPLIER = multiplierNumber;
}
```

For example:

Assuming that at block 8239999, the `wagPerBlock` is set to 10 $WAG per block, pool 0 `allocPoint` is set to 300, `totalAllocPoint` is set to 9605, and `BONUS_MULTIPLIER` is a constant.

Block	Action
8239999	All pools' rewards are updated
8249999	A new pool is added using the `add()` function, causing the `totalAllocPoint` to be changed from 9605 to 10000
8259999	The pools' rewards are updated once again

From current logic, the total rewards allocated to the pool 0 during block 8239999 to block 8259999 is equal to 6,000.00 $WAG (`totalRewardBlock`) calculated using the following equation:

```
(pool 0 allocPoint / totalAllocPoint) * wagPerBlock * totalRewardBlock = reward
(300 / 10,000) * 10 * 20,000 = 6,000
```

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

- 0.3123 $WAG per block, from block 8239999 to block 8249999, with a proportion of 300/9,605 = 3,123.37 $WAG
- 0.3000 $WAG per block, from block 8249999 to block 8259999, with a proportion of 300/10,000 = 3,000.00 $WAG

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

In addition, if the `BONUS_MULTIPLIER` is changed, the reward will be miscalculated as the example above.

5.13.2. Remediation

Inspex suggests removing `_withUpdate` variable in the `set()` and the `add()` functions and always calling the `massUpdatePools()` and the `updateStakingPool()` function before updating `totalAllocPoint` variable. This also includes calling the `updateMultiplier()` function to include the `massUpdatePools()` and the `updateStakingPool()` function before updating `BONUS_MULTIPLIER` as shown in the following example:

MasterChef.sol

```
// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
```

```
function add(
    uint256 _allocPoint,
    ERC20 _lpToken,
  ) public onlyOwner {
    massUpdatePools();
    updateStakingPool();
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({ lpToken: _lpToken, allocPoint: _allocPoint, lastRewardBlock:
lastRewardBlock, accWagPerShare: 0 })
    );
}
```

MasterChef.sol

```
// Update the given pool's CAKE allocation point. Can only be called by the
owner.
function set(
    uint256 _pid,
    uint256 _allocPoint,
  ) public onlyOwner {
    massUpdatePools();
    updateStakingPool();
    uint256 prevAllocPoint = poolInfo[_pid].allocPoint;
    poolInfo[_pid].allocPoint = _allocPoint;
    if (prevAllocPoint != _allocPoint) {
        totalAllocPoint = totalAllocPoint.sub(prevAllocPoint).add(_allocPoint);
    }
}
```

MasterChef.sol

```
function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
    massUpdatePools();
    updateStakingPool();
    BONUS_MULTIPLIER = multiplierNumber;
}
```

5.14. Improper Reward Calculation (Duplicate lpToken)

ID	IDX-014
Target	MasterChef
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Medium The reward of the pool that has the same staking token as the reward token will be lower than what it should be depended on the deposit amount. **Likelihood:** Low It is very unlikely that the platform will add multiple pools with the same staking tokens due to the nature of the `MasterChef` contract.
Status	Resolved Waggy Finance team has resolved this issue as suggested by validating the LP address to ensure that it is not added `0a130ebb6fda808576d0fdf4dbddc5b45d3f0b77`.

5.14.1. Description

In the `MasterChef` contract, a new staking pool can be added using the `add()` function. The staking token for the new pool is defined using the `lpToken` variable; however, there is no additional checking whether the `lpToken` is the same as the existing pool or not.

MasterChef.sol

```
// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(
    uint256 _allocPoint,
    ERC20 _lpToken,
    bool _withUpdate
  ) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({ lpToken: _lpToken, allocPoint: _allocPoint, lastRewardBlock: lastRewardBlock, accWagPerShare: 0 })
```

```
    );
    updateStakingPool();
}
```

When the `lpToken` is the same token as the `lpToken` in the added pools, the reward calculation for that pool in the `updatePool()` function can be incorrect. This is because the current balance of the `lpToken` in the contract is used in the calculation of the reward as in line 174 and 183.

MasterChef.sol

```
168 // Update reward variables of the given pool to be up-to-date.
169 function updatePool(uint256 _pid) public {
170     PoolInfo storage pool = poolInfo[_pid];
171     if (block.number <= pool.lastRewardBlock) {
172         return;
173     }
174     uint256 lpSupply = pool.lpToken.balanceOf(address(this));
175     if (lpSupply == 0) {
176         pool.lastRewardBlock = block.number;
177         return;
178     }
179     uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
180     uint256 wagReward =
multiplier.mul(wagPerBlock).mul(pool.allocPoint).div(totalAllocPoint);
181     wag.mint(devaddr, wagReward.div(10));
182     wag.mint(address(this), wagReward);
183     pool.accWagPerShare =
pool.accWagPerShare.add(wagReward.mul(1e12).div(lpSupply));
184     pool.lastRewardBlock = block.number;
185 }
```

This means by using the same `lpToken` for multiple pools, the reward minted to the `MasterChef` contract will be less than what it should be due to the addition of the `lpToken` amount (`lpSupply`).

5.14.2. Remediation

Inspex suggests checking the value of the `lpToken` in the `add()` function to prevent the pool with the same staking token as the existing pool from being added, for example:

MasterChef.sol

```
mapping(address => bool) public existLp;
// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up
if you do.
function add(
    uint256 _allocPoint,
    ERC20 _lpToken,
```

```
    bool _withUpdate
  ) public onlyOwner {
    require(!existLp[address(_lpToken)]);
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number :
startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(
        PoolInfo({ lpToken: _lpToken, allocPoint: _allocPoint, lastRewardBlock:
lastRewardBlock, accWagPerShare: 0 })
    );
    existLp[address(_lpToken)] = true;
    updateStakingPool();
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

5.15. Design Flaw in massUpdatePool() Function

ID	IDX-015
Target	MasterChef
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Low **Impact:** Medium The `massUpdatePool()` function will eventually be unusable due to excessive gas usage. **Likelihood:** Low It is very unlikely that the `poolInfo` size will be raised until the `massUpdatePool()` function is eventually unusable.
Status	Resolved Waggy Finance team has resolved this issue as suggested by implementing a function to remove unnecessary pools in commit `e4042c5c76cccb0474beb9a1f3cffc0a6c50ac4c`.

5.15.1. Description

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

MasterChef.sol

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

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

5.15.2. Remediation

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

5.16. Design Flaw in Reward Distribution Model

ID	IDX-016
Target	GasStation
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Low **Impact:** Low The reward that the platform provides through the `refillPool()` function will be able to claim instantly, which normally distributes linearly according to the business design. This causes the platform to have less liquidity from what it should be. **Likelihood:** Medium It is likely that the attackers will perform this attack since there is no restriction. However, it requires a decent capital in order to gain a worthiness reward.
Status	Resolved Waggy Finance team has resolved this issue as suggested by changing the reward distribution model in commit `2f5e5e423a7c49d47e3e04763756dad8557720dc`.

5.16.1. Description

In `GasStation` contract, it allows the platform users to stake an NFTs through the `stake()` function and receive the reward through the `claim()` function, similarly to the `MasterChef` contract concept.

GasStation.sol

```
function stake(address _nftAddress, uint256 _tokenId) external {
  PoolInfo storage pool = poolInfo;
  UserInfo storage user = userInfo[msg.sender];
  // claim reward before new staking
  if (user.weights > 0) {
    uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
      pool.lpToken.transfer(address(msg.sender), pending);
    }
  }

  WNFT wnft = WNFT(_nftAddress);
  uint256 weight = wnft.getWeight();
  require(weight > 0, "can't stake");
  wnft.safeTransferFrom(msg.sender, address(this), _tokenId);

```

```
  if (weight > 0) {
    pool.supply = pool.supply.add(weight);
    user.nftStake[_nftAddress] = user.nftStake[_nftAddress].add(1);
    user.weights = user.weights.add(weight);
  }
  user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

  emit Stake(msg.sender, _nftAddress, _tokenId, weight);
}
```

The difference is that the reward will not be minted directly from the contract itself, but rather anyone who wants to distribute reward, in this case the platform, through the `refillPool()` function. The `pool.accWagPerShare` will be updated, resulting in a new pending claimable reward for the users.

GasStation.sol

```
function refillPool(uint256 _amount) public {
  PoolInfo storage pool = poolInfo;
  pool.lpToken.transferFrom(msg.sender, address(this), _amount);
  pool.accWagPerShare =
pool.accWagPerShare.add(_amount.mul(1e12).div(pool.supply));
  pool.lastRewardBlock = block.number;
}
```

However, with the current design, the attacker can monitor the mempool for a `refillPool()` function transaction and execute the `stake()` function to stake an NFTs before `pool.accWagPerShare` is updated, causing the instant claimable reward. The attacker can further call the `unStake()` function to remove an NFTs afterward. Hence, the attacker can claim the reward freely, taking all advantages from the other platform users.

GasStation.sol

```
function unStake(address _nftAddress, uint256 _tokenId) external {
  PoolInfo storage pool = poolInfo;
  UserInfo storage user = userInfo[msg.sender];
  require(user.nftStake[_nftAddress] > 0, "No NFT Stake");
  // Claim reward before unstake
  uint256 pending =
user.weights.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
  if (pending > 0) {
    pool.lpToken.transfer(address(msg.sender), pending);
  }

  WNFT wnft = WNFT(_nftAddress);
  uint256 weight = wnft.getWeight();
  user.nftStake[_nftAddress] = user.nftStake[_nftAddress].sub(1);
  user.weights = user.weights.sub(weight);
```

```
    wnft.safeTransferFrom(address(this), msg.sender, _tokenId);

    user.rewardDebt = user.weights.mul(pool.accWagPerShare).div(1e12);

    emit UnStake(msg.sender, _nftAddress, _tokenId, weight);
}
```

5.16.2. Remediation

Inspex suggests adding the emission rate mechanism to be fair to all platform users and prevent the loss of liquidity from what it should be for the `GasStation` contracts. To elaborate about the emission rate, it is a mechanism that has the end time duration for the reward distribution, this means the time of the user who stakes to the pool will be considered as the factor to claim the reward.

5.17. Design Flaw in claimAll() Function

ID	IDX-017
Target	MasterChef
Category	General Smart Contract Vulnerability
CWE	CWE-400: Uncontrolled Resource Consumption
Risk	**Severity:** Very Low **Impact:** Low The `claimAll()` function will eventually be unusable due to excessive gas usage. However, the users can claim each pool individually. **Likelihood:** Low It is very unlikely that the `poolInfo` size will be raised until the `claimAll()` function is eventually unusable.
Status	Resolved Waggy Finance team has resolved this issue as suggested by implementing function to remove unnecessary pool in commit `e4042c5c76cccb0474beb9a1f3cffc0a6c50ac4c`.

5.17.1. Description

The `claimAll()` function allows the caller to claim rewards for all existing pools in the `MasterChef` contract.

MasterChef.sol

```
function claimAll() public {
    for (uint256 index = 0; index < poolInfo.length; index++) {
        claim(index);
    }
}
```

MasterChef.sol

```
function claim(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 pending =
user.amount.mul(pool.accWagPerShare).div(1e12).sub(user.rewardDebt);
    if (pending > 0) {
        user.rewardDebt = user.amount.mul(pool.accWagPerShare).div(1e12);
        safeWagTransfer(msg.sender, pending);
    }
}
```

However, the `claimAll()` function iterates to all pools, regardless if the users already deposit to that pool or not. Hence, with the current design as described in `IDX-00x Design Flaw in massUpdatePool() Function`, if new pools continue to be added to this contract, the `poolInfo.length` will continue to grow and this function will eventually be unusable due to excessive gas usage.

5.17.2. Remediation

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

5.18. Insufficient Logging for Privileged Functions

ID	IDX-018
Target	AvatarNFT GasStation MasterChef FeeCalculator MerchantMultiToken RewardCalculator WaggyToken WNativeRelayer BlackListUser Validator
Category	General 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	Resolved Waggy Finance team has resolved this issue as suggested by emitting events for the execution of privileged functions `6c50cad0083f09442f45b02f2b1b80a159456645`.

5.18.1. Description

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

For example, the owner can add the admin address in the `adminRole[_admin]` state by executing the `setAdmin()` function in the `Validator` contract, and no events are emitted.

Validator.sol

```
function setAdmin(address _admin, bool _isAdmin) public onlyOwner {
    adminRole[_admin] = _isAdmin;
}
```

The privileged functions without sufficient logging are as follows:

File	Contract	Function
AvatarNFT.sol (L:34)	AvatarNFT	setPrice()
AvatarNFT.sol (L:51)	AvatarNFT	claim()
AvatarNFT.sol (L:60)	AvatarNFT	setBaseURI()
GasStation.sol (L:86)	GasStation	setAdmin()
MasterChef.sol (L:79)	MasterChef	updateMultiplier()
MasterChef.sol (L:89)	MasterChef	add()
MasterChef.sol (L:106)	MasterChef	set()
MasterChef.sol (L:135)	MasterChef	setLockRewardPercent()
MasterChef.sol (L:300)	MasterChef	dev()
FeeCalculator.sol (L:37)	FeeCalculator	updateFeeRate()
RewardCalculator.sol (L:27)	RewardCalculator	updateRewardRate()
WaggyToken.sol (L:67)	WaggyToken	setEndReleaseBlock()
WaggyToken.sol (L:88)	WaggyToken	setMinter()
WaggyToken.sol (L:97)	WaggyToken	revokeRoles()
WNativeRelayer.sol (L:25)	WNativeRelayer	setCallerOk()
BlackListUser.sol (L:47)	BlackListUser	setUserStatus()
BlackListUser.sol (L:57)	BlackListUser	setAdmins()
Validator.sol (L:111)	Validator	setAdmin()
Validator.sol (L:115)	Validator	setMinPercent()
Validator.sol (L:119)	Validator	setMaxPercent()
MerchantMultiToken.sol (L:137)	MerchantMultiToken	setValidator()
MerchantMultiToken.sol (L:141)	MerchantMultiToken	setAdmins()
MerchantMultiToken.sol (L:150)	MerchantMultiToken	setWNativeRelayer()
MerchantMultiToken.sol (L:154)	MerchantMultiToken	setWBNB()
MerchantMultiToken.sol (L:158)	MerchantMultiToken	revokeRoles()

MerchantMultiToken.sol (L:517)	MerchantMultiToken	setBlackList()
MerchantMultiToken.sol (L:526)	MerchantMultiToken	ownerClaimToken()
MerchantMultiToken.sol (L:531)	MerchantMultiToken	updateRewardCalculator()
MerchantMultiToken.sol (L:536)	MerchantMultiToken	updateFeeCalculator()

5.18.2. Remediation

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

Validator.sol

```
event SetAdmin(address user, address admin);
function setAdmin(address _admin, bool _isAdmin) public onlyOwner {
    adminRole[_admin] = _isAdmin;
    emit SetAdmin(_admin, _isAdmin);
}
```

5.19. Inexplicit Solidity Compiler Version

ID	IDX-019
Target	AvatarNFT GasStation MasterChef FeeCalculator MerchantMultiToken RewardCalculator WaggyToken WNativeRelayer BlackListUser Validator
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Waggy Finance team has resolved this issue as suggested by changing the solidity version to the latest version in commit `1575bf26fded4c518b1d154e3d61f0dc3fcd6922`.

5.19.1. 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.

AvatarNFT.sol

```
//SPDX-License-Identifier: Unlicense
/*
*/
pragma solidity ^0.8.0;
```

The following table contains all contracts that use an inexplicit solidity compiler version.

Target	Version
AvatarNFT.sol (L:10)	^0.8.0
GasStation.sol (L:10)	^0.8.0

MasterChef.sol (L:10)	^0.8.0
FeeCalculator.sol (L:11)	^0.8.0
MerchantMultiToken.sol (L:10)	^0.8.0
RewardCalculator.sol (L:10)	^0.8.0
WaggyToken.sol (L:10)	^0.8.0
WNativeRelayer.sol (L:2)	^0.8.0
BlackListUser.sol (L:10)	^0.8.0
Validator.sol (L:10)	^0.8.0

5.19.2. 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 v0.8 is v0.8.11 https://github.com/ethereum/solidity/releases/tag/v0.8.11.

AvatarNFT.sol

```
//SPDX-License-Identifier: Unlicense
/*
*/
pragma solidity 0.8.11;
```

5.20. Improper Function Visibility

ID	IDX-020
Target	Validator AvatarNFT GasStation MasterChef FeeCalculator RewardCalculator WaggyToken MerchantMultiToken
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Waggy Finance team has resolved this issue as suggested by changing all the function visibilities to external in commit `2f5e5e423a7c49d47e3e04763756dad8557720dc`.

5.20.1. Description

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

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

AvatarNFT.sol

```
// Mint all NFT on deploy and keep data for treading
function mint(address _receiver) public payable {
  require(msg.value == nftPrice, "Price missmatch");
  require(userOwnerTokenId[msg.sender] == 0, "Maximun to mint");
  uint256 newItemId = _tokenIds.current();
  _mint(_receiver, newItemId);
  _tokenIds.increment();

  userOwnerTokenId[msg.sender] = newItemId;

  emit Mint(msg.sender, newItemId);
}
```

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

Target	Function
Validator.sol (L:111)	setAdmin()
Validator.sol (L:123)	addCase()
Validator.sol (L:157)	getUserResultInCase()
Validator.sol (L:171)	userClaimReward()
Validator.sol (L:190)	userCanClaimReward()
Validator.sol (L:210)	encode()
Validator.sol (L:217)	decideByAdmin()
Validator.sol (L:252)	setCaseStatusDone()
Validator.sol (L:260)	evaluate()
AvatarNFT.sol (L:39)	mint()
GasStation.sol (L:86)	setAdmin()
GasStation.sol (L:97)	refillPool()
MasterChef.sol (L:79)	updateMultiplier()
MasterChef.sol (L:89)	add()
MasterChef.sol (L:106)	set()
MasterChef.sol (L:187)	deposit()
MasterChef.sol (L:224)	withdraw()
MasterChef.sol (L:244)	enterStaking()
MasterChef.sol (L:263)	leaveStaking()
MasterChef.sol (L:282)	emergencyWithdraw()
MasterChef.sol (L:300)	dev()
FeeCalculator.sol (L:37)	updateFeeRate()
RewardCalculator.sol (L:27)	updateRewardRate()
MerchantMultiToken.sol (L:150)	setWNativeRelayer()

MerchantMultiToken.sol (L:166)	depositNative()
MerchantMultiToken.sol (L:175)	deposit()
MerchantMultiToken.sol (L:184)	withdrawNative()
MerchantMultiToken.sol (L:198)	withdraw()
MerchantMultiToken.sol (L:213)	approveTransaction()
MerchantMultiToken.sol (L:243)	fetchTransactionApproved()
MerchantMultiToken.sol (L:257)	cancelTransactionSeller()
MerchantMultiToken.sol (L:279)	releaseTokenBySeller()
MerchantMultiToken.sol (L:521)	getFeeCollector()
MerchantMultiToken.sol (L:526)	ownerClaimToken()
MerchantMultiToken.sol (L:531)	updateRewardCalculator()
MerchantMultiToken.sol (L:536)	updateFeeCalculator()
MerchantMultiToken.sol (L:540)	getBuyerTransaction()

5.20.2. Remediation

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

AvatarNFT.sol

```
// Mint all NFT on deploy and keep data for treading
function mint(address _receiver) external payable {
  require(msg.value == nftPrice, "Price missmatch");
  require(userOwnerTokenId[msg.sender] == 0, "Maximun to mint");
  uint256 newItemId = _tokenIds.current();
  _mint(_receiver, newItemId);
  _tokenIds.increment();

  userOwnerTokenId[msg.sender] = newItemId;

  emit Mint(msg.sender, newItemId);
}
```

Please note that the remediation for other issues are not yet applied in the examples above.

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

inspex

CYBERSECURITY PROFESSIONAL SERVICE