Ratoken

Smart Contract Audit Report
Prepared for Ratoken

Date Issued: Sep 13, 2021
Project ID: AUDIT2021023
Version: v1.0
Confidentiality Level: Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021023
Version	v1.0
Client	Ratoken
Project	Ratoken
Auditor(s)	Weerawat Pawanawiwat Peeraphut Punsuwan
Author	Weerawat Pawanawiwat
Reviewer	Pongsakorn Sommalai
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Sep 13, 2021	Full report	Weerawat Pawanawiwat

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Ratoken, Inspex team conducted an audit to verify the security posture of the Ratoken smart contract on Sep 9, 2021. During the audit, Inspex team examined the smart contract and the overall operation within the scope to understand the overview of the Ratoken smart contract. 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 1 high, 1 medium, 2 low, 1 very low, and 2 info-severity issues. With the project team's prompt response, 1 high, 1 medium, 1 low, 1 very low, and 2 info-severity issues were resolved in the reassessment, while 1 low-severity issue was acknowledged by the team. Therefore, Inspex trusts that Ratoken smart contract has sufficient protections to be safe for public use. However, in the long run, Inspex suggests resolving all issues found in this report.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

Ratoken is a 3D MMORPG-GameFi on the Binance Smart Chain Network with a deflationary token integrated into the game.

Ratoken is a deflationary token with reflective reward distribution and automatic liquidity providing mechanisms to provide benefits to the holders. It is also used as the main in-game currency for Ratoken, a 3D-MMORPG game with the same name.

Scope Information:

Project Name	Ratoken
Website	https://ratoken.app/
Smart Contract Type	Ethereum Smart Contract
Chain	Binance Smart Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Sep 9, 2021
Reassessment Date	Sep 13, 2021

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

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

2.2. Scope

The following smart contract was audited and reassessed by Inspex in detail:

Initial Audit:

Contract	Location (URL)
Token	https://bscscan.com/address/0x3A79F0Adda5Ed957F2A01A68DbAd99a8Db9bCBB5

Reassessment:

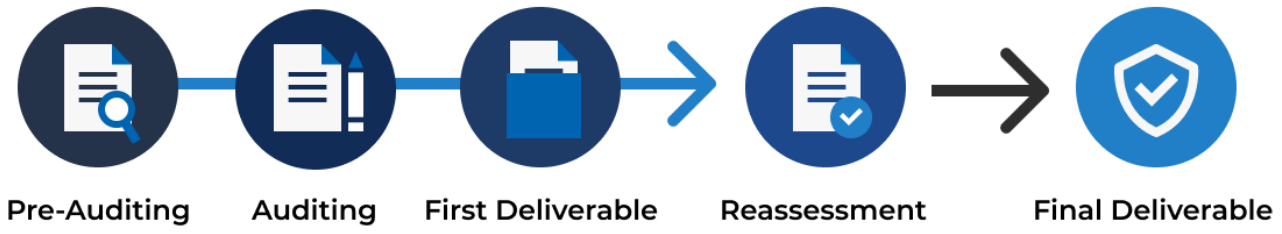
Contract	Location (URL)
Token	https://bscscan.com/address/0x43c612590Ad7Ac3f5fc217Bf71487B49A034E195

The assessment scope covers only the in-scope smart contract and the smart contracts that they are inherited from.

3. Methodology

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

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



3.1. Test Categories

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

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

3.2. Audit Items

The following audit items were checked during the auditing activity.

General
Reentrancy Attack
Integer Overflows and Underflows
Unchecked Return Values for Low-Level Calls
Bad Randomness
Transaction Ordering Dependence
Time Manipulation
Short Address Attack
Outdated Compiler Version
Use of Known Vulnerable Component
Deprecated Solidity Features
Use of Deprecated Component
Loop with High Gas Consumption
Unauthorized Self-destruct
Redundant Fallback Function
Advanced
Business Logic Flaw
Ownership Takeover
Broken Access Control
Broken Authentication
Upgradable Without Timelock
Improper Kill-Switch Mechanism
Improper Front-end Integration
Insecure Smart Contract Initiation

Denial of Service
Improper Oracle Usage
Memory Corruption
Best Practice
Use of Variadic Byte Array
Implicit Compiler Version
Implicit Visibility Level
Implicit Type Inference
Function Declaration Inconsistency
Token API Violation
Best Practices Violation

3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

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

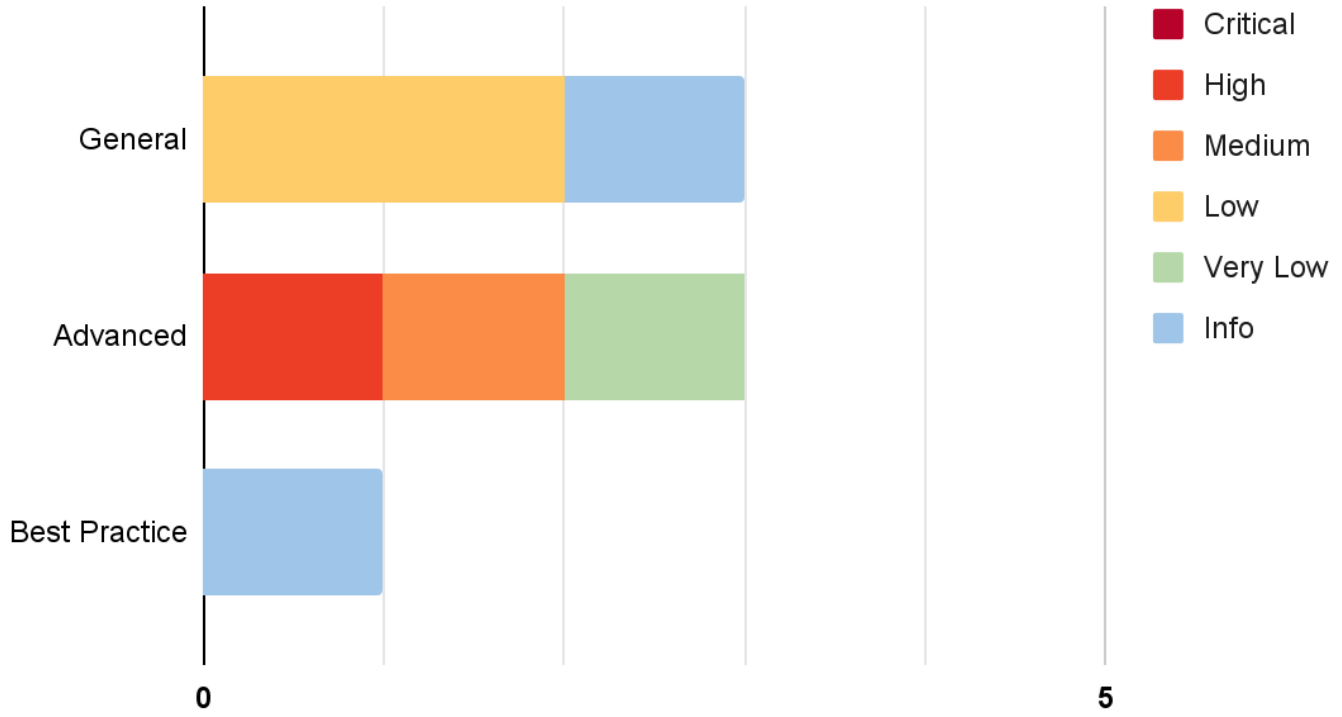
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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

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 7 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	Improper Ownership Locking Mechanism	Advanced	**High**	Resolved
IDX-002	Improper Balance Calculation	Advanced	**Medium**	Resolved
IDX-003	Centralized Control of State Variable	General	**Low**	Acknowledged
IDX-004	Division Before Multiplication	General	**Low**	Resolved
IDX-005	Insufficient Logging for Privileged Functions	Advanced	**Very Low**	Resolved
IDX-006	Unchecked Return Value of ERC20 Transfer	General	**Info**	Resolved
IDX-007	Improper Function Visibility	Best Practice	**Info**	Resolved

5. Detailed Findings Information

5.1. Improper Ownership Locking Mechanism

ID	IDX-001
Target	Token
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** High **Impact:** High The previous contract owner can reclaim the ownership of the contract when the ownership is transferred to another address or renounced, giving that address the ability to perform privileged actions again. **Likelihood:** Medium Only the contract owner can perform this action; however, there is no restriction to prevent the owner from abusing this flaw.
Status	Resolved Ratoken team has resolved this issue as suggested and deployed the fixed version to 0x43c612590Ad7Ac3f5fc217Bf71487B49A034E195 on BSC mainnet.

5.1.1. Description

The `Token` contract inherits a custom `Ownable` contract with an ownership locking mechanism. The contract owner can execute the `lock()` function to temporarily transfer the ownership of the contract to zero address for a duration specified in the `time` parameter. The address of the owner is saved in the `_previousOwner` state variable to be used later.

Token.sol

```
522 //Locks the contract for owner for the amount of time provided
523 function lock(uint256 time) public virtual onlyOwner {
524     _previousOwner = _owner;
525     _owner = address(0);
526     _lockTime = block.timestamp + time;
527     emit OwnershipTransferred(_owner, address(0));
528 }
```

After a duration of `time` has passed, the address stored in the `_previousOwner` state variable can call the `unlock()` function to reclaim the ownership of the contract.

Token.sol

```
530 //Unlocks the contract for owner when _lockTime is exceeds
531 function unlock() public virtual {
532     require(_previousOwner == msg.sender, "You don't have permission to unlock
    the token contract");
533     require(block.timestamp > _lockTime , "Contract is locked until 7 days");
534     emit OwnershipTransferred(_owner, _previousOwner);
535     _owner = _previousOwner;
536 }
```

However, the `_previousOwner` state variable is not cleared in the `unlock()` function, this allows the address stored in the `_previousOwner` state variable to reclaim the ownership even when the ownership has been renounced or transferred to another address.

As an example, assuming the contract owner is wallet A, the attacking steps can be as follows:

1. Wallet A executes the `lock()` function to store the address of wallet A in the `_previousOwner` state variable.
2. Wallet A executes the `unlock()` function to reclaim the contract ownership.
3. Wallet A renounces the ownership or transfers the ownership to another address. At the current state, the users will see that the owner of the contract is not wallet A, and may trust that privileged functions cannot be executed by wallet A anymore.
4. Wallet A executes the `unlock()` function, transferring the ownership back to wallet A.

5.1.2. Remediation

Inspex suggests clearing the `_previousOwner` state variable at the end of the `unlock()` function to prevent it from being executed multiple times, for example:

Token.sol

```
530 //Unlocks the contract for owner when _lockTime is exceeds
531 function unlock() public virtual {
532     require(_previousOwner == msg.sender, "You don't have permission to unlock
533 the token contract");
534     require(block.timestamp > _lockTime , "Contract is locked until 7 days");
535     emit OwnershipTransferred(_owner, _previousOwner);
536     _owner = _previousOwner;
537     _previousOwner = address(0);
    }
```

5.2. Improper Balance Calculation

ID	IDX-002
Target	Token
Category	Advanced Smart Contract Vulnerability
CWE	CWE-840: Business Logic Errors
Risk	**Severity:** Medium **Impact:** Medium The balance of a previously excluded address can be inflated by using the `includeInReward()` function, which is unfair to the token holders. **Likelihood:** Medium The exclusion and inclusion of an address can only be done by the contract owner; however, it is possible that the contract owner will perform this action since it is profitable.
Status	Resolved Ratoken team has resolved this issue by removing the `includeInReward()` function and deployed the fixed version to 0x43c612590Ad7Ac3f5fc217Bf71487B49A034E195 on BSC mainnet.

5.2.1. Description

The `Token` contract has a reward mechanism based on $RFI to distribute the fees to the balance of the holders. The holders are separated into two groups, those who are included in the reward distribution and those who are excluded. The balance between two groups are calculated separately.

Token.sol

```
835 function balanceOf(address account) public view override returns (uint256) {
836     if (_isExcluded[account]) return _tOwned[account];
837     return tokenFromReflection(_rOwned[account]);
838 }
```

The contract owner can execute the `excludeFromReward()` function to exclude an address from gaining the fee reward.

Token.sol

```
904 function excludeFromReward(address account) public onlyOwner() {
905     require(!_isExcluded[account], "Account is already excluded from reward");
906     if(_rOwned[account] > 0) {
907         _tOwned[account] = tokenFromReflection(_rOwned[account]);
908     }
909     _isExcluded[account] = true;
```

```
910         _excluded.push(account);
911     }
```

When an address is excluded, its balances will be excluded from the supply used in the token balance calculation.

Token.sol

```
1015 function _getCurrentSupply() private view returns(uint256, uint256) {
1016     uint256 rSupply = _rTotal;
1017     uint256 tSupply = _tTotal;
1018     for (uint256 i = 0; i < _excluded.length; i++) {
1019         if (_rOwned[_excluded[i]] > rSupply || _tOwned[_excluded[i]] > tSupply)
1020 return (_rTotal, _tTotal);
1021         rSupply = rSupply.sub(_rOwned[_excluded[i]]);
1022         tSupply = tSupply.sub(_tOwned[_excluded[i]]);
1023     }
1024     if (rSupply < _rTotal.div(_tTotal)) return (_rTotal, _tTotal);
1025     return (rSupply, tSupply);
     }
```

The contract owner can also use the `includeInReward()` function to reinclude an address for it to gain the reward.

Token.sol

```
913 function includeInReward(address account) external onlyOwner() {
914     require(_isExcluded[account], "Already excluded");
915     for (uint256 i = 0; i < _excluded.length; i++) {
916         if (_excluded[i] == account) {
917             _excluded[i] = _excluded[_excluded.length - 1];
918             _tOwned[account] = 0;
919             _isExcluded[account] = false;
920             _excluded.pop();
921             break;
922         }
923     }
924 }
```

However, when that address is included again, the `_rOwned` balance of that address is not recalculated from the value of `_tOwned`, causing this address to gain the reward from the period when it is excluded.

5.2.2. Remediation

Inspex suggest calculating the `_rOwned` balance of the address on the reinclusion of it to the reward using the current rate, for example:

Token.sol

```
913 function includeInReward(address account) external onlyOwner() {
914     require(_isExcluded[account], "Already excluded");
915     for (uint256 i = 0; i < _excluded.length; i++) {
916         if (_excluded[i] == account) {
917             _excluded[i] = _excluded[_excluded.length - 1];
918             uint256 currentRate =  _getRate();
919             _rOwned[account] = _tOwned[account].mul(currentRate);
920             _tOwned[account] = 0;
921             _isExcluded[account] = false;
922             _excluded.pop();
923             break;
924         }
925     }
926 }
```

5.3. Centralized Control of State Variable

ID	IDX-003
Target	Token
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Low **Impact:** Low The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done by the owner; however, the changes are limited by fixed values in the smart contracts.
Status	**Acknowledged** Ratoken team has acknowledged this issue. The team has decided to remove the `includeInReward()`, `setAllFeePercent()`, `setMaxTxPercent()`, and `setMaxWalletPercent()` functions that have high impact to the users and deployed the modified version to 0x43c612590Ad7Ac3f5fc217Bf71487B49A034E195 on BSC mainnet. Therefore, Inspex has adjusted the impact and severity of this issue from Medium to Low.

5.3.1. 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.

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:

File	Contract	Function	Inherit From	Modifier
Token.sol (L:503)	Token	renounceOwnership()	Ownable	onlyOwner
Token.sol (L:512)	Token	transferOwnership()	Ownable	onlyOwner
Token.sol (L:523)	Token	lock()	Ownable	onlyOwner
Token.sol (L:904)	Token	excludeFromReward()	-	onlyOwner
Token.sol (L:913)	Token	includeInReward()	-	onlyOwner

Token.sol (L:927)	Token	excludeFromFee()	-	onlyOwner
Token.sol (L:931)	Token	includeInFee()	-	onlyOwner
Token.sol (L:935)	Token	setAllFeePercent()	-	onlyOwner
Token.sol (L:952)	Token	setBuybackUpperLimit()	-	onlyOwner
Token.sol (L:956)	Token	setMaxTxPercent()	-	onlyOwner
Token.sol (L:963)	Token	setMaxWalletPercent()	-	onlyOwner
Token.sol (L:970)	Token	setSwapAndLiquifyEnabled()	-	onlyOwner
Token.sol (L:975)	Token	setFeeWallet()	-	onlyOwner

5.3.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a `Timelock` contract to delay the changes for a sufficient amount of time, e.g., 24 hours

5.4. Division Before Multiplication

ID	IDX-004
Target	Token
Category	General Smart Contract Vulnerability
CWE	CWE-682: Incorrect Calculation
Risk	**Severity:** Low **Impact:** Low The rounding error can cause the fees to be slightly miscalculated. **Likelihood:** Medium It is likely that the balance divided by each fee can result in a decimal value.
Status	Resolved Ratoken team has resolved this issue as suggested and deployed the fixed version to 0x43c612590Ad7Ac3f5fc217Bf71487B49A034E195 on BSC mainnet.

5.4.1. Description

Solidity supports only integer values but not floating point values. The division of integers can result in a value with decimal points, which will be rounded off. This rounding error can cause the calculation to be different from what it should be, especially when that value is later multiplied with another value.

For example, in line 1153, the `spentAmount` is calculated by dividing `contractTokenBalance` with `totFee`, before multiplying it by `_burnFee`. The rounding error caused by the division is amplified in the multiplication, and can increase the amount of miscalculation.

Token.sol

```
1146 function swapAndLiquify(uint256 contractTokenBalance) private lockTheSwap {
1147     //This needs to be distributed among burn, wallet and liquidity
1148     //burn
1149     uint8 totFee  = _burnFee + _walletFee + _liquidityFee + _buybackFee;
1150     uint256 spentAmount = 0;
1151     uint256 totSpentAmount = 0;
1152     if(_burnFee != 0){
1153         spentAmount  = contractTokenBalance.div(totFee).mul(_burnFee);
1154         _tokenTransferNoFee(address(this), dead, spentAmount);
1155         totSpentAmount = spentAmount;
1156     }
```

The lines of code with division before multiplication are as follows:

- Token.sol (L:1153)

- Token.sol (L:1159)
- Token.sol (L:1165)

5.4.2. Remediation

Inspex suggests modifying the affected lines of code to perform multiplication before division, for example:

Token.sol

```
1146 function swapAndLiquify(uint256 contractTokenBalance) private lockTheSwap {
1147     //This needs to be distributed among burn, wallet and liquidity
1148     //burn
1149     uint8 totFee  = _burnFee + _walletFee + _liquidityFee + _buybackFee;
1150     uint256 spentAmount = 0;
1151     uint256 totSpentAmount = 0;
1152     if(_burnFee != 0){
1153         spentAmount  = contractTokenBalance.mul(_burnFee).div(totFee);
1154         _tokenTransferNoFee(address(this), dead, spentAmount);
1155         totSpentAmount = spentAmount;
1156     }
1157 
1158     if(_walletFee != 0){
1159         spentAmount = contractTokenBalance.mul(_walletFee).div(totFee);
1160         _tokenTransferNoFee(address(this), feeWallet, spentAmount);
1161         totSpentAmount = totSpentAmount + spentAmount;
1162     }
1163 
1164     if(_buybackFee != 0){
1165         spentAmount = contractTokenBalance.mul(_buybackFee).div(totFee);
1166         swapTokensForBNB(spentAmount);
1167         totSpentAmount = totSpentAmount + spentAmount;
1168     }
1169 
1170     if(_liquidityFee != 0){
1171         contractTokenBalance = contractTokenBalance.sub(totSpentAmount);
1172 
1173         // split the contract balance into halves
1174         uint256 half = contractTokenBalance.div(2);
1175         uint256 otherHalf = contractTokenBalance.sub(half);
1176 
1177         // capture the contract's current ETH balance.
1178         // this is so that we can capture exactly the amount of ETH that the
1179         // swap creates, and not make the liquidity event include any ETH that
1180         // has been manually sent to the contract
1181         uint256 initialBalance = address(this).balance;
1182 
1183         // swap tokens for ETH
1184         swapTokensForBNB(half); // <- this breaks the ETH -> HATE swap when
swap+liquify is triggered
```

```
1185
1186            // how much ETH did we just swap into?
1187            uint256 newBalance = address(this).balance.sub(initialBalance);
1188
1189            // add liquidity to uniswap
1190            addLiquidity(otherHalf, newBalance);
1191
1192            emit SwapAndLiquify(half, newBalance, otherHalf);
1193        }
1194
1195    }
```

5.5. Insufficient Logging for Privileged Functions

ID	IDX-005
Target	Token
Category	Advanced Smart Contract Vulnerability
CWE	CWE-778: Insufficient Logging
Risk	**Severity:** Very Low **Impact:** Low Privileged functions' executions cannot be monitored easily by the users. **Likelihood:** Low It is not likely that the execution of the privileged functions will be a malicious action.
Status	Resolved Ratoken team has resolved this issue as suggested and deployed the fixed version to 0x43c612590Ad7Ac3f5fc217Bf71487B49A034E195 on BSC mainnet. Please note that the `includeInReward()`, `setAllFeePercent()`, `setMaxTxPercent()`, and `setMaxWalletPercent()` functions have been removed.

5.5.1. Description

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

For example, the owner can modify the fees by executing `setAllFeePercent()` function in the `Token` contract, and no event is emitted.

The privileged functions without sufficient logging are as follows:

File	Contract	Function	Modifier
Token.sol (L:904)	Token	excludeFromReward()	onlyOwner
Token.sol (L:913)	Token	includeInReward()	onlyOwner
Token.sol (L:927)	Token	excludeFromFee()	onlyOwner
Token.sol (L:931)	Token	includeInFee()	onlyOwner
Token.sol (L:935)	Token	setAllFeePercent()	onlyOwner
Token.sol (L:952)	Token	setBuybackUpperLimit()	onlyOwner

Token.sol (L:956)	Token	setMaxTxPercent()	onlyOwner
Token.sol (L:963)	Token	setMaxWalletPercent()	onlyOwner

5.5.2. Remediation

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

Token.sol

```
904 event ExcludeFromReward(address account);
905 function excludeFromReward(address account) public onlyOwner() {
906     require(!_isExcluded[account], "Account is already excluded from reward");
907     if(_rOwned[account] > 0) {
908         _tOwned[account] = tokenFromReflection(_rOwned[account]);
909     }
910     _isExcluded[account] = true;
911     _excluded.push(account);
912     emit ExcludeFromReward(account);
913 }
```

5.6. Unchecked Return Value of ERC20 Transfer

ID	IDX-006
Target	Token
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standard
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Ratoken team has resolved this issue as suggested and deployed the fixed version to 0x43c612590Ad7Ac3f5fc217Bf71487B49A034E195 on BSC mainnet.

5.6.1. Description

ERC20 tokens can be implemented in multiple ways, allowing the execution of failed `transfer()` and `transferFrom()` functions by returning `false` instead of reverting when the invalid transfer amount occurs.

In the `Token` contract, the `recoverBEP20()` function can be used to transfer other tokens in the contract to the contract owner.

Token.sol

```
1328 function recoverBEP20(address tokenAddress, uint256 tokenAmount) public
     onlyOwner {
1329     // do not allow recovering self token
1330     require(tokenAddress != address(this), "Self withdraw");
1331     IERC20(tokenAddress).transfer(owner(), tokenAmount);
1332 }
```

The return value of the `transfer()` function is not checked, so the transfer transactions of tokens that return `false` on failure will not be reverted.

However, there's no impact in this case since this function is only used for the tokens mistakenly transferred to the `Token` contract, and is not related to the users' balances.

5.6.2. Remediation

Inspex suggests replacing the `transfer()` function with `safeTransfer()` function from OpenZeppelin's `SafeERC20` library, for example:

Token.sol

```
1328 function recoverBEP20(address tokenAddress, uint256 tokenAmount) public
     onlyOwner {
1329     // do not allow recovering self token
1330     require(tokenAddress != address(this), "Self withdraw");
1331     IERC20(tokenAddress).safeTransfer(owner(), tokenAmount);
1332 }
```

5.7. Improper Function Visibility

ID	IDX-007
Target	Token
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved Ratoken team has resolved this issue as suggested and deployed the fixed version to 0x43c612590Ad7Ac3f5fc217Bf71487B49A034E195 on BSC mainnet. Please note that the `includeInReward()`, `setAllFeePercent()`, `setMaxTxPercent()`, and `setMaxWalletPercent()` functions have been removed.

5.7.1. Description

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

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

Token.sol

```
840 function transfer(address recipient, uint256 amount) public override returns
    (bool) {
841     _transfer(_msgSender(), recipient, amount);
842     return true;
843 }
```

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

File	Contract	Function	Inherit From
Token.sol (L:503)	Token	renounceOwnership()	Ownable
Token.sol (L:512)	Token	transferOwnership()	Ownable
Token.sol (L:523)	Token	lock()	Ownable

Token.sol (L:531)	Token	unlock()	Ownable
Token.sol (L:840)	Token	transfer()	-
Token.sol (L:845)	Token	allowance()	-
Token.sol (L:849)	Token	approve()	-
Token.sol (L:854)	Token	transferFrom()	-
Token.sol (L:860)	Token	increaseAllowance()	-
Token.sol (L:865)	Token	decreaseAllowance()	-
Token.sol (L:878)	Token	deliver()	-
Token.sol (L:904)	Token	excludeFromReward()	-
Token.sol (L:913)	Token	includeInReward()	-
Token.sol (L:927)	Token	excludeFromFee()	-
Token.sol (L:931)	Token	includeInFee()	-
Token.sol (L:935)	Token	setAllFeePercent()	-
Token.sol (L:907)	Token	setSwapAndLiquifyEnabled()	-
Token.sol (L:1328)	Token	recoverBEP20()	-

5.7.2. Remediation

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

Token.sol

```
840 function transfer(address recipient, uint256 amount) external override returns
    (bool) {
841     _transfer(_msgSender(), recipient, amount);
842     return true;
843 }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

6.2. References

[1] “OWASP Risk Rating Methodology.” [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE