Token

Smart Contract Audit Report
Prepared for iAM

Date Issued:	Jan 19, 2022
Project ID:	AUDIT2021042
Version:	v1.0
Confidentiality Level:	Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2021042
Version	v1.0
Client	iAM
Project	Token
Auditor(s)	Suvicha Buakhom Peeraphut Punsuwan Darunphop Pengkumta
Author(s)	Natsasit Jirathammanuwat
Reviewer	Weerawat Pawanawiwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
1.0	Jan 19, 2022	Full report	Natsasit Jirathammanuwat

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 iAM, Inspex team conducted an audit to verify the security posture of the Token smart contracts between Jan 5, 2022 and Jan 6, 2022. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of Token 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 2 medium, 2 very low and 1 info-severity issues. With the project team's prompt response, 1 medium, 2 very low and 1 info-severity issues were resolved in the reassessment, while 1 medium issue was acknowledged by the team. However, Inspex suggests resolving all issues found in this report to reduce the risk and improve the security level of the smart contracts.

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

Token (EventToken & MainToken) are the ERC20 token contracts that are used for participation in iAM events. The users can use tokens for voting on the election or purchasing merchandise on the iAM platform.

Scope Information:

Project Name	Token
Website	https://www.bnk48.com/
Smart Contract Type	Ethereum Smart Contract
Chain	TKX Chain
Programming Language	Solidity

Audit Information:

Audit Method	Whitebox
Audit Date	Jan 5, 2022 - Jan 6, 2022
Reassessment Date	Jan 17, 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: 0f8bb7a04c439286a5dffa223c3b30265acb4b5c)

Contract	Location (URL)
EventToken	https://github.com/inspex-archive/iAM-Admin-Poll-Token/blob/0f8bb7a04c/EventToken.sol
MainToken	https://github.com/inspex-archive/iAM-Admin-Poll-Token/blob/0f8bb7a04c/MainToken.sol

Please note that the iAM smart contracts were initially in iAM's private repository. The files have been uploaded to Inspex's archive repository for public access.

Reassessment:

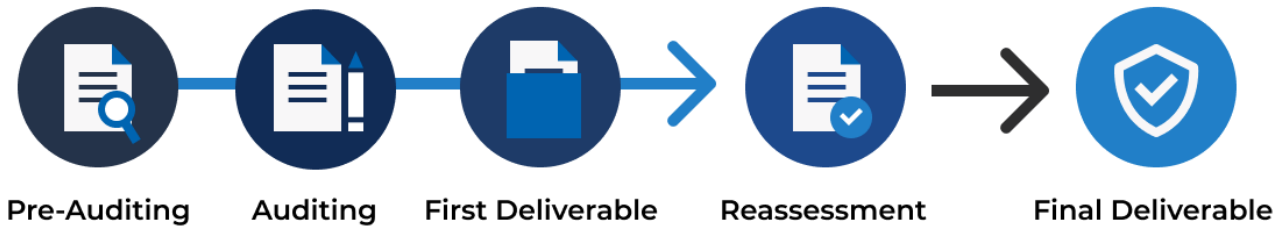
Contract	Location (URL)
EventToken	https://scan.tokenx.finance/address/0xcB682b97CAF58290494aC8E076e08333D5AFa21e/contracts
MainToken	https://scan.tokenx.finance/address/0xA992ad80Fa6136702382123Ae717890Bc587491d/contracts

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
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[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

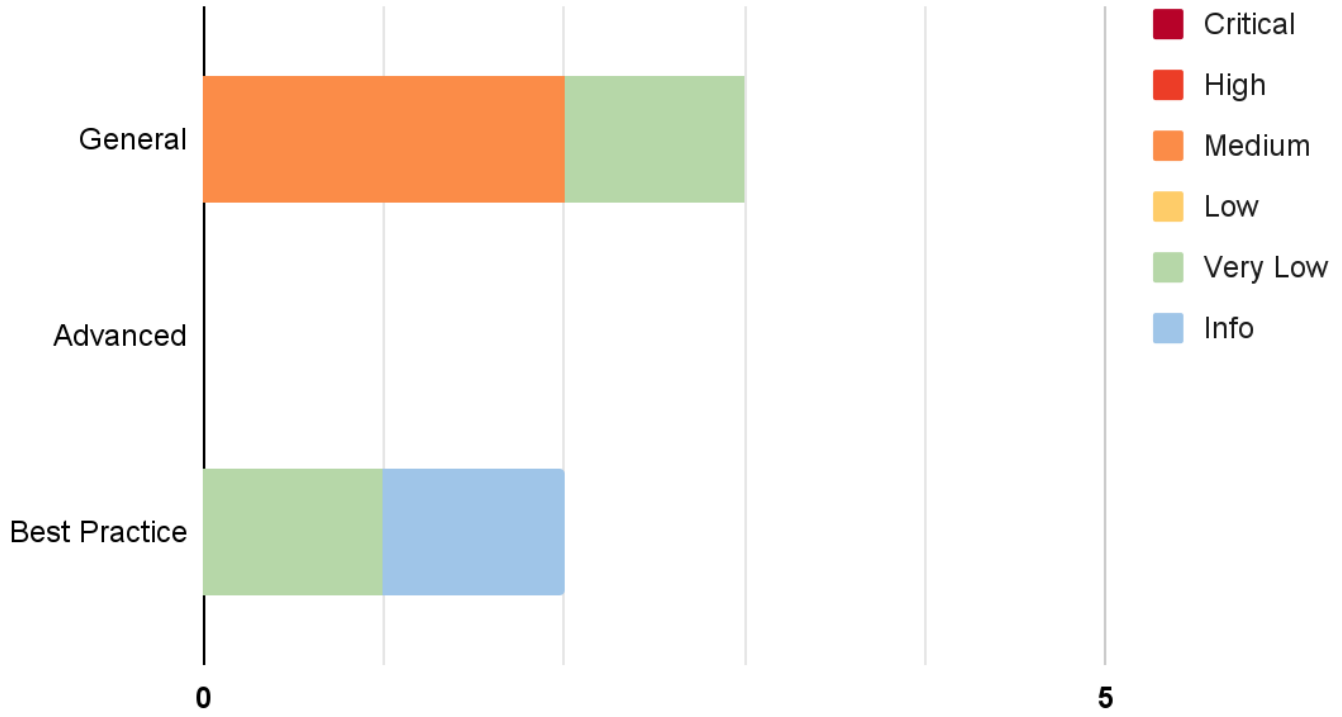
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

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 5 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	Arbitrarily Tokens Transfer	General	**Medium**	**Acknowledged**
IDX-002	Arbitrarily Tokens Minting	General	**Medium**	**Resolved**
IDX-003	Outdated Solidity Compiler Version	General	**Very Low**	**Resolved**
IDX-004	Unnecessary Assert Statement	Best Practice	**Very Low**	**Resolved**
IDX-005	Gas Usage Optimization	Best Practice	**Info**	**Resolved**

5. Detailed Findings Information

5.1. Arbitrarily Tokens Transfer

ID	IDX-001
Target	EventToken MainToken
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Medium **Impact:** High Admin users can transfer tokens from any address to any address. The tokens that users received can be transferred to other addresses. So, users can lose the possession of the tokens by the privilege of the owners. **Likelihood:** Low Only the owners can use this function. This function requires a dual control process to perform the action.
Status	**Acknowledged** The iAM team has acknowledged this issue and stated that the `adminTransfer()` function is used for emergency cases only, and at least two authorized management persons are required to execute the `adminTransfer()` function. However, there are still some risks for the users as any two of the authorized management persons can arbitrarily transfer users' tokens to any wallet.

5.1.1. Description

Admin users can transfer tokens from any address arbitrarily. The `adminTransfer()` function was implemented by using `_transfer()` function directly, without having any restrictions for transferring. As a result, admin users can transfer tokens from anyone to anyone, arbitrarily.

At line 35 of `EventToken.sol` and at line 29 of `MainToken.sol`, the `_transfer()` function is used without any prior condition to restrict the transfer.

EventToken.sol

```
30 function adminTransfer(
31     address sender,
32     address recipient,
33     uint256 amount
34 ) external onlyOwner returns (bool) {
35     _transfer(sender, recipient, amount);
```

```
36
37      emit AdminTransfer(sender, recipient, amount);
38      return true;
39  }
```

MainToken.sol

```
24  function adminTransfer(
25      address sender,
26      address recipient,
27      uint256 amount
28  ) external onlyOwner returns (bool) {
29      _transfer(sender, recipient, amount);
30
31      emit AdminTransfer(sender, recipient, amount);
32      return true;
33  }
```

However, the ownership will be transferred to the `BigOwner` contract when this contract is deployed. The `BigOwner` contract requires dual control to call the `adminTransfer()` (1 admin of the `BigOwner` creates pending of the `adminTransfer()` function transaction and requires another admin to execute it).

MainToken.sol

```
18  constructor(
19      string memory _name,
20      string memory _symbol,
21      uint256 _initialToken,
22      IAdminManage _adminManage,
23      address _mintTo,
24      address _bigOwner
25  ) ERC20(_name, _symbol) Adminnable(_adminManage) {
26      _mint(_mintTo, _initialToken);
27      transferOwnership(_bigOwner);
28  }
```

5.1.2. Remediation

Inspex suggests removing the `adminTransfer()` function. To transfer the tokens, the normal `transfer()` function can be used instead.

5.2. Arbitrarily Token Minting

ID	IDX-002
Target	EventToken
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Medium **Impact:** Medium The contract admin can arbitrarily mint the affected tokens, leading to unfair token distribution. **Likelihood:** Medium Only the contract admin can execute the `mint()` function.
Status	Resolved The iAM team has resolved this issue by removing the `mint()` function from the `EventToken` contract.

5.2.1. Description

According to the business requirement, the contract admin can unlimitedly mint tokens by performing the `mint()` function in the `EventToken` contract, so the users should accept or be notified before the token minting is effective as shown in the following source code:

EventToken.sol

```
49 function mint(address to, uint256 amount) external onlyAdmin {
50     _mint(to, amount);
51
52     assert(_totalMint + amount >= _totalBurn);
53     _totalMint = _totalMint + amount;
54     emit Mint(to, amount);
55 }
```

5.2.2. Remediation

In the ideal case, the token must not be minted freely. Thus, Inspex suggests removing the `mint()` function from the `EventToken` contract.

However, in the case that the `mint()` function is required, Inspex suggests mitigating this issue by limiting the use of this function via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a Timelock contract to delay the minting action for a reasonable amount of time

5.3. Outdated Solidity Compiler Version

ID	IDX-003
Target	EventToken MainToken
Category	General Smart Contract Vulnerability
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Very Low **Impact:** Low From the list of known Solidity bugs, direct impact cannot be caused from those bugs themselves. **Likelihood:** Low From the list of known Solidity bugs, it is very unlikely that those bugs would affect these smart contracts.
Status	**Resolved** The iAM team has resolved this issue by upgrading the Solidity compiler to 0.8.9 which currently does not have any known vulnerability.

5.3.1. Description

The solidity compiler versions declared in the smart contracts were outdated. These versions have publicly known inherent bugs[2] that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

EventToken.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity =0.8.4;
```

The following table contains all targets which the outdated compiler version is declared.

Contract	Version
EventToken	=0.8.4
MainToken	=0.8.4

5.3.2. Remediation

Inspex suggests upgrading the solidity compiler to the latest stable version[3]. At the time of the audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.11.

EventToken.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity =0.8.11;
```

Please note that some dependencies that are imported in each contract require a specific compiler version, please verify their compatibility before upgrading them.

5.4. Unnecessary Assert Statement

ID	IDX-004
Target	EventToken MainToken
Category	Advanced Smart Contract Vulnerability
CWE	CWE-1164: Irrelevant Code
Risk	**Severity:** Very Low **Impact:** Low The owner of the `EventToken` contract cannot mint a low amount of token when the `_totalBurn` amount is high, and functions that use the unnecessary assert statement will consume more gas. **Likelihood:** Low It is very unlikely that the `_totalBurn` amount will be higher than the sum of the `_totalMint` and minting amount because new minting always increases the `_totalMint` amount.
Status	Resolved The iAM team has resolved this issue by removing the `mint()` function from the `EventToken` contract and the `_beforeTokenTransfer()` function from the `EventToken` and the `MainToken` contracts which contain unnecessary assert statements.

5.4.1. Description

Some condition values of the assert statements in the `EventToken` and the `MainToken` contracts are always `true`. So, it is not necessary to check and can be removed to reduce the gas usage.

There is an exception case in line 52 of `EventToken.sol`. The assert statement in this point is breaking the functionality of the `mint()` function. The sum of `_totalMint` and minting amount needs to be more than or equal the `_totalBurn` amount. So, if some of the initial tokens were burnt before minting, it will require the new minting amount to exceed the `_totalBurn` amount.

EventToken.sol

```
49 function mint(address to, uint256 amount) external onlyAdmin {
50     _mint(to, amount);
51
52     assert(_totalMint + amount >= _totalBurn);
53     _totalMint = _totalMint + amount;
54     emit Mint(to, amount);
55 }
```

In line 63 of `EventToken.sol`, and line 45 of `MainToken.sol`, are unnecessary having the assert statements, because the result of the condition is always `true`.

EventToken.sol

```
57 function _beforeTokenTransfer(
58     address from,
59     address to,
60     uint256 amount
61 ) internal override {
62     if (to == address(0)) {
63         assert(_totalBurn + amount >= _totalBurn);
64 
65         _totalBurn = _totalBurn + amount;
66         emit Burn(from, amount);
67     }
68 }
```

MainToken.sol

```
39 function _beforeTokenTransfer(
40     address from,
41     address to,
42     uint256 amount
43 ) internal override {
44     if (to == address(0)) {
45         assert(_totalBurn + amount >= _totalBurn);
46 
47         _totalBurn = _totalBurn + amount;
48         emit Burn(from, amount);
49     }
50 }
```

5.4.2. Remediation

Inspex suggests removing the affected assert statements that are unnecessary, for example:

MainToken.sol

```
39 function _beforeTokenTransfer(
40     address from,
41     address to,
42     uint256 amount
43 ) internal override {
44     if (to == address(0)) {
45         _totalBurn = _totalBurn + amount;
46         emit Burn(from, amount);
47     }
48 }
```

5.5. Gas Usage Optimization

ID	IDX-005
Target	EventToken MainToken
Category	Smart Contract Best Practice
CWE	CWE-1164: Irrelevant Code
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The iAM team has resolved this issue by removing the `_beforeTokenTransfer()` function from the `EventToken` and the `MainToken` contracts.

5.5.1. Description

The `_beforeTokenTransfer()` function is called every time the transfer happens.

@openzeppelin/contracts/token/ERC20/ERC20.sol

```
219 function _transfer(
220     address sender,
221     address recipient,
222     uint256 amount
223 ) internal virtual {
224     require(sender != address(0), "ERC20: transfer from the zero address");
225     require(recipient != address(0), "ERC20: transfer to the zero address");
226
227     _beforeTokenTransfer(sender, recipient, amount);
228
229     uint256 senderBalance = _balances[sender];
230     require(senderBalance >= amount, "ERC20: transfer amount exceeds balance");
231     unchecked {
232         _balances[sender] = senderBalance - amount;
233     }
234     _balances[recipient] += amount;
235
236     emit Transfer(sender, recipient, amount);
237
238     _afterTokenTransfer(sender, recipient, amount);
239 }
```

The `_beforeTokenTransfer()` function checks the receiver address of transfer, making the transfer transaction consume more gas.

EventToken.sol

```
57 function _beforeTokenTransfer(
58     address from,
59     address to,
60     uint256 amount
61 ) internal override {
62     if (to == address(0)) {
63         assert(_totalBurn + amount >= _totalBurn);
64
65         _totalBurn = _totalBurn + amount;
66         emit Burn(from, amount);
67     }
68 }
```

The `_transfer()` function is the main function that is used frequently. The higher gas usage in this function increases the gas consumption for the users of this contract.

5.5.2. Remediation

Inspex suggest removing the `_beforeTokenTransfer()` function from the `EventToken` and the `MainToken` contracts and calculating the total burn amount as follows:

For the `EventToken` contract, declare the `initialToken` state to store the initial minting value, remove the `_totalBurn` state variable, and calculate the total burnt amount using `_totalMint + initialToken - totalSupply`, for example:

EventToken.sol

```
18 uint256 private initialToken;
19
20 constructor(
21     string memory _name,
22     string memory _symbol,
23     uint256 _initialToken,
24     IAdminManage _adminManage,
25     address _mintTo,
26     address _bigOwner
27 ) ERC20(_name, _symbol) Adminnable(_adminManage) {
28     _mint(_mintTo, _initialToken);
29     initialToken = _initialToken;
30     transferOwnership(_bigOwner);
31 }
32
33 function totalBurn() external view returns (uint256) {
```

```
34      return _totalMint + initialToken - totalSupply;
35  }
```

For the `MainToken` contract, declare the `initialToken` state variable and set the `initialToken` state with the `_initialToken` value in the constructor. That value should be used in the calculation in `totalBurn()` function that calculate the total burnt amount using `initialToken - totalSupply`, for example:

MainToken.sol

```
14  uint256 private initialToken;
15
16  constructor(
17      string memory _name,
18      string memory _symbol,
19      uint256 _initialToken,
20      address _bigOwner
21  ) ERC20(_name, _symbol) {
22      _mint(msg.sender, _initialToken);
23      initialToken = _initialToken;
24      transferOwnership(_bigOwner);
25  }
26
27  function totalBurn() external view returns (uint256) {
28      return initialToken - totalSupply;
29  }
```

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: 18-Jan-2022]

[2] “List of Known Bugs — Solidity 0.8.12 documentation” [Online]. Available: https://docs.soliditylang.org/en/latest/bugs.html. [Accessed: 18-Jan-2022]

[3] “Releases · ethereum/solidity” [Online]. Available: https://github.com/ethereum/solidity/releases. [Accessed: 18-Jan-2022]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE