



KICKICO Token Audit



Foreword

Status of a vulnerability or security issue

Clarity is a rare commodity. That is why for the convenience of both the client and the reader, we have introduced a system of marking vulnerabilities and security issues we discover during our security audits.

No issue

Let's start with an ideal case. If an identified security imperfection bears no impact on the security of our client, we mark it with the label.

✓ Fixed

The fixed security issues get the label that informs those reading our public report that the flaws in question should no longer be worried about.

Addressed

In case a client addresses an issue in another way (e.g., by updating the information in the technical papers and specification) we put a nice tag right in front of it.

By design

If an issue is planned to be addressed in the future or it is a feature that has been designed this way deliberately, we put the "By design" label.

Although the issues marked "Fixed" and "By design" are no threat, we still list them to provide the most detailed and up-to-date information for the client and the reader.

Severity levels

We also rank the magnitude of the risk a vulnerability or security issue pose. For this purpose, we use 4 "severity levels" namely:

1. Minor

2. Medium

3. Major

4. Critical

More details about the ranking system as well as the description of the severity levels can be found in [Appendix 1. Terminology](#).

TABLE OF CONTENTS

01. INTRODUCTION

1. Source code

Security assessment methodology

Auditors

02. SUMMARY

03. GENERAL ISSUES

1. require check typo

✓ Fixed

2. Token migration issue

By design

3. Feature request

✓ Fixed

4. Pause and Burn trigger methods simplification

✓ Fixed

5. Frozen tokens storage issue

✓ Fixed

6. High risk of changeOwner implementation

✓ Fixed

7. Lack of Transfer event for the _mintfrozen method

By design

8. Redundant require

✓ Fixed

9. ApproveAndCall has to return bool

✓ Fixed

10. Transfer event usage for mint and burn operations

By design

11. make accountBalance, _frozen_balanceOf public

By design

12. destroy method doesn't destroy frozen_balance which leads to complex issues

✓ Fixed

13. Remove unused methods

✓ Fixed

14. Separate role for meltTokens function

✓ Fixed

15. Wrong modifiers

✓ Fixed

CONCLUSION

APPENDIX 1. TERMINOLOGY

1. Severity

01. Introduction

1 Source code

Provided as a zipped file.

2 Security assessment methodology

The code of a smart contract has been automatically and manually scanned for known vulnerabilities and logic errors that may cause security threats. The conformity of requirements (e.g., White Paper) and practical implementation has been reviewed as well. More information on the used methodology can be found [here](#).

3 Auditors

1. [Alexey Pertsev](#)

2. [Roman Storm](#)

02. Summary

Below, you can find a table with all the discovered bugs and security issues listed.

Security issue	Severity
destroy method doesn't destroy frozen_balance which leads to complex issues	CRITICAL
Separate role for meltTokens function	
Wrong modifiers	MAJOR
require check typo	
Token migration issue	MEDIUM
Feature request	
High risk of changeOwner implementation	
Transfer event usage for mint and burn operations	
Pause and Burn trigger methods simplification	MINOR
Frozen tokens storage issue	
Lack of Transfer event for the _mintfrozen method	
Redundant require	
ApproveAndCall has to return bool	
make accountBalance, _frozen_balanceOf public	
Remove unused methods	

03. General issues

1

require check typo

Severity: MEDIUM

```
function transferFrozenToken(address from, address to, uint256 amount) public onlyOwner returns (bool) {
    require(from != address(0), "ERC20: transfer from the zero address");
    → require(from != address(0), "ERC20: transfer to the zero address");
    require(frozenbalances[from].balanceOf(from) >= amount, "ERC20: transfer frozen tokens: balance of send

    frozenbalances[from].sub(amount);
    frozenbalances[to].add(amount, now);

    emit FrozenTransfer(from, to, amount);
    return true;
}
```

Duplicated `require`. It's supposed to be equal to `!= address(0)`

Status:

✓ Fixed

2

Token migration issue

Severity: MEDIUM

According to the current architecture, the KICKICO team will take a snapshot of old KICKICO token holders and propagate their balances to a new token. Since there are 18.5k addresses in there, it will cost at least 6-16 ETH.

We recommend making use of the `approveAndCall` function of the old contract to allow users to move their tokens themselves. KICKICO can also reward users with some tokens for their work.

This strategy enables identifying active users and save those 6-16 ETH mentioned above.

Status:

By design

The team decided to keep its airdrop strategy for the sake of user convenience

3

Feature request

Severity: **MEDIUM**

Consider adding the `approveAndCall` functionality. It will come in useful in future integrations with DEX and DeFi services.

Status:

✓ Fixed

4

Pause and Burn trigger methods simplification

Severity: **MINOR**

Consider using

```
1 function pauseTrigger() public onlyOwner {  
2     paused = !paused;  
3 }
```

instead of

```
1 function pauseTrigger(bool _paused) public onlyOwner {  
2     paused = _paused;  
3 }
```

It will simplify the owner's UX. The same logic is also applicable to the `burnTrigger` method.

Status:


✓ Fixed

5

Frozen tokens storage issue

Severity: **MINOR**

In the current implementation, `_frozenbalanceOf` shows how many tokens out of the user's total balance are frozen, which requires to change `_balancesmapping` in all the methods that work with frozen tokens (`_mintfrozen`, `_melt`, `_burnFrozen`, etc.).

```
function _mintfrozen(address account, uint256 amount) internal {  
    require(account != address(0), "ERC20: mint frozen to the zero address");  
    require(account != address(this), "ERC20: mint frozen to the contract address");  
    require(amount > 0, "ERC20: mint frozen amount should be > 0");  
  
    _totalSupply = _totalSupply.add(amount);  
     _balances[account] = _balances[account].add(amount);  
    emit Transfer(address(this), account, amount);  
  
    _frozen_add(account, amount);  
    emit MintFrozen(account, amount);  
}
```

To avoid multiple recalculations, you can design `_frozen_balanceOf` as “how many additional frozen tokens a user has”. So the `balanceOf` would be:

```
1 function balanceOf(address account) public view returns (uint256) {  
2     return _balances[account].add(_frozen_balanceOf(account));  
3 }
```

and `accountBalance` is no longer necessary.

Status:

✓ Fixed

6

High risk of changeOwner implementation

Severity: **MEDIUM**

The current `changeOwner` implementation does not check `newOwner` in any way:

```
1 function changeOwner (address newOwner) public onlyOwner {  
2     owner = newOwner;  
3 }
```

Recommendations:

1. Consider using the `transferOwnership` + `claimOwnership` pattern to avoid possible issues.

```
1 address public pendingOwner;  
2 /**  
3  * @dev Modifier throws if called by any account other than the pendingOwner.  
4  */  
5 modifier onlyPendingOwner() {  
6     require(msg.sender == pendingOwner);  
7     _;  
8 }  
9 /**  
10  * @dev Allows the current owner to set the pendingOwner address.  
11  * @param newOwner The address to transfer ownership to.  
12  */  
13 function transferOwnership(address newOwner) onlyOwner {  
14     pendingOwner = newOwner;  
15 }  
16 /**  
17  * @dev Allows the pendingOwner address to finalize the transfer.  
18  */  
19 function claimOwnership() onlyPendingOwner {  
20     OwnershipTransferred(owner, pendingOwner);  
21     owner = pendingOwner;  
22     pendingOwner = 0x0;  
23 }
```

Status:

✓ Fixed

7

Lack of Transfer event for the _mintfrozen methodSeverity: **MINOR**

The _mintfrozen method emits the Transfer(address(this), account, amount); event, which is supposed to notify about the minting process. But it does not emit the Transfer(account, address(this), amount); event to indicate that freezing takes place (like in freezeTokensmethod).

Status:

By design The team decided to leave it as is

8

Redundant requireSeverity: **MINOR**

The require:

```
1 | require(_frozen_balanceOf(from) >= amount, «ERC20: transfer frozen tokens: balance of sender should be >= amount»);
```

is redundant. If `_frozen_balanceOf(from)` is less than than the `amount`, the contract will revert the transaction due to SafeMath overflow exception (see `_frozen_sub implementation`).

Status:

✓ Fixed

9

ApproveAndCall has to return boolSeverity: **MINOR**

It's advisable to have explicit return as boolean.:

```
1 | return spender.receiveApproval(msg.sender, _value, address(this), _extraData);
```

Also, make the contract interface capitalize the first letter in `TokenRecipient`.

Status:

✓ Fixed

10

Transfer event usage for mint and burn operationsSeverity: **MEDIUM**

During the `mint` and `burn` operations, contract should emit the `Transfer` event from and to `address(0)` respectively.

Status:

By design

The team decided to leave it as is (the project requirements)

11

make `accountBalance`, `_frozen_balanceOf` publicSeverity: **MINOR**

In order to know correct transferable balance, users should be able to see that information

```
1 | function accountBalance(address account) public view returns (uint256);
```

Status:

By design

`accountBalance` no longer exists. `_frozen_balanceOf` is still private according to requirements.

12

destroy method doesn't destroy frozen_balance which leads to complex issuesSeverity: **CRITICAL**

1. mint 10 tokens to user1.
2. freeze 5 tokens to user1
3. destroy 10 tokens to user1

Now we have a few problems:

`accountBalance` reverts because `SafeMath` reverts due to `assert(0 - 5)` for unsigned integer is not possible. It breaks the whole calculation of token balances. It is also impossible to call `destroyFrozen` even though there are still 5 frozen tokens, because there is 0 balance. Lets say we mint 10 tokens again to user1. Now, what `totalSupply` should be? 5 frozen + 10 new = 15 tokens. But, we only have 10 now. If frozen tokens should be a part of `totalSupply`, it is a critical issue.

Recommendations:

1. add `accountBalance` check to the `_burn` function.

Status:

✓ Fixed

13

Remove unused methods

Severity: **MINOR**

Those methods are not used anywhere.

```
1      mapping (address => bool) private _whitelisted;
2      modifier onlyWhitelisted() {
3          require(_whitelisted[msg.sender] == true, "caller does not have the
Whitelisted role");
4      };
5  }
6
7      function addWhitelisted(address account) public onlyOwner {
8          _whitelisted[account] = true;
9      }
10
11     function removeWhitelisted(address account) public onlyOwner {
12         _whitelisted[account] = false;
13     }
```

Status:

✓ Fixed

14

Separate role for meltTokens function

Severity: **MAJOR**

We recommend adding a separate role for the `meltTokens` function to reduce a possible impact of compromising the melting service the team has developed for the project.

Status:

✓ Fixed

15

Wrong modifiers

Severity: **MAJOR**

After implementing separate roles for minters, melters, and owner, `mintFrozenTokens` still has the `onlyOwner` modifier, which is supposed to be `onlyMinter`.

The mint has the `onlyOwner` modifier as well. Consider changing it to `onlyMinter` as well.

Status:

✓ Fixed

Conclusion

During the audit, the experts have discovered 1 **Critical** issue along with other 14 security issues that fall into the **Major**, **Medium**, and **Minor** severity groups.

To properly mitigate the discovered security imperfections and better protection of the token, Peppersec has offered a range of recommendations that, if applied, will strengthen the architecture and better the code style. Many of these security measures have already been implemented by the KICKICO security team, thus boosting the resilience of the token; others were qualified as “By design” and do not require fixes.

Appendix 1. Terminology

1 Severity

Severity is the category that described the magnitude of an issue.

		Severity		
Impact	Major	Medium	Major	Critical
	Medium	Minor	Medium	Major
	Minor	None	Minor	Medium
		Minor	Medium	Major
Likelihood				

MINOR

Minor issues are generally subjective in their nature or potentially associated with the topics like “best practices” or “readability”. As a rule, minor issues do not indicate an actual problem or bug in the code. The maintainers should use their own judgment as to whether addressing these issues will improve the codebase.

MEDIUM

Medium issues are generally objective in their nature but do not represent any actual bugs or security problems. These issues should be addressed unless there is an apparent reason not to.

MAJOR

Major issues are things like bugs or vulnerabilities. These issues may be unexploitable directly or may require a certain condition to arise to be exploited. If unaddressed, these issues are likely to cause problems with the operation of the contract or lead to situations which make the system exploitable.

CRITICAL

Critical issues are directly exploitable bugs or security vulnerabilities. If unaddressed, these issues are likely or guaranteed to cause major problems and ultimately a full failure in the operations of the contract.

About Us

Worried about the security of your project? You're on the right way! The second step is to find a team of seasoned cybersecurity experts who will make it impenetrable. And you've just come to the right place.

PepperSec is a group of whitehat hackers seasoned by many-year experience and have a deep understanding of the modern Internet technologies. We're ready to battle for the security of your project.

LET'S KEEP IN TOUCH



peppersec.com



hello@peppersec.com



[Github](#)