PUBLIC REPORT

KICKICO Token Audit

peppersec

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.

Acknowledged

If an issue is planned to be addressed in the future, it gets the tag, and a client clearly sees what is yet to be done.

Although the issues marked "Fixed" and "Acknowledged" 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

The source code was provided as a zip file. We uploaded the first two versions to our **github repo** in favor of reader usability. The final version was deployed to mainnet at **0xc12d1c73ee7dc3615ba4e37e4abfdbddfa38907e** address.

2 Security assessment methodology

The code of a smart contract is scanned manually and automatically for known vulnerabilities and logic errors that can lead to potential security threats. The conformity of requirements (e.g., White Paper) and practical implementations are reviewed as well on a consistent basis. More about the methodology can be found **here**.

3 Auditors

1. **Alexey Pertsev**
2. **Roman Storm**

02. Critical issues

1 Frozen and active token balance desynchronization

Severity: CRITICAL

The destroy method doesn't destroy frozen_balance, which can lead to complex issues.

Example:

- mint 10 tokens to a user. Now the user has 10 active tokens.
- Smart contract owner freezes 5 tokens of the user by calling freezeTokens method so the user still has 10 tokens, but 5 of them are frozen.
- Smart contract owner burns 10 tokens of the user by calling the destroy method. The method burns tokens from _balances mapping while the frozen_balances mapping is left unchanged.

At this point, the balanceOf view returns 0. The _frozen_balanceOf returns 5, and the accountBalance reverts because SafeMath calculates 0 - 5 to an error for unsigned integers and breaks the calculation of token balances.

It's also impossible to call destroyFrozen even though there are 5 frozen tokens remaining, since the current balance is 0.

Let's mint another 10 tokens to the user. So what should totalSupply be? In theory it is 5 frozen tokens + 10 new ones totalling to 15. However, we only have 10.

Recommendation:

Add accountBalance check to the _burn function.

Status:

✓ Fixed

03. Major issues

1 Lack of access separation

Severity: MAJOR

The **meltTokens** function has `onlyOwner` access modifier, but it's used by a separate melting service. It's not neccessary for the service to have access to other privileged functions like `mint`, `destroy`, etc. Therefore, if the melting service is compromised, the system may have greater exposure.

Recommendation:

Consider separating the role for the `meltTokens` function

Status:

✓ Fixed

2 Wrong modifiers usage

Severity: MAJOR

The second version has separate roles for minter, melter and owner (see "Lack of access separation" above). But the **mintFrozenTokens** and **mint** methods still have `onlyOwner` modifier, which is supposed to be `onlyMinter`.

Recommendation:

Consider changing the modifiers.

Status:

✓ Fixed

04. Medium 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;
}
```

Duplicate require from present when it should be:

```
1    require(to != address(0), “ERC20: transfer to the zero address”)
```

The bug was discovered before the audit during the cost estimation, so the first version provided already contained the **fix**.

Recommendation:

Consider changing the require expression.

Status:

✓ Fixed

2 Expensive token migration

Severity: MEDIUM

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

Recommendation:

Consider using the approveAndCall function of the old contract to allow users to move their tokens themselves. KICKICO team can also reward users with tokens for their work. This strategy allows to determine active users and save 6-16 ETH we mentioned above.

Status:

✓ Fixed

3 Risky implementation of changeOwner functionality

Severity: **MEDIUM**

Current **changeOwner** implementation does not validate newOwner:

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

Recommendations:

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

```
address public pendingOwner;

/**
* @dev Modifier throws if called by any account other than the pendingOwner.
*/
modifier onlyPendingOwner() {
    require(msg.sender == pendingOwner);
    _;
}

/**
* @dev Allows the current owner to set the pendingOwner address.
* @param newOwner The address to transfer ownership to.
*/
function transferOwnership(address newOwner) onlyOwner {
    pendingOwner = newOwner;
}


/**
* @dev Allows the pendingOwner address to finalize the transfer.
*/
function claimOwnership() onlyPendingOwner {
    OwnershipTransferred(owner, pendingOwner);
    owner = pendingOwner;
    pendingOwner = 0x0;
}
```

Status:

✓ Fixed

4 Transfer event usage for mint and burn operations

Severity: **MEDIUM**

During **mint** and **burn** operations the contract should emit `Transfer` event from and to `address(0)` respectively. This is not a ERC20 standard requirement but is one of best practices that majority of Ethereum projects utilize.

Recommendations:

Consider changing the event arguments.

Status:

Acknowledged Left as is (project requirements).

05. Minor issues

1 The approveAndCall feature

Severity: **MINOR**

The current version does not have a way to send tokens to a smart contract and trigger some functionality of it. According to modern standards, the approveAndCall function may be implemented for this purpose. It will also be useful for future integrations with DEX and DeFi services.

Recommendation:

Consider adding approveAndCall funtionality.

Status:

✓ Fixed

2 pauseTrigger and burnTrigger methods simplification

Severity: **MINOR**

The token contract has functionality that stops and starts token transfer and destruction. These methods can be optimized to better the interface.

Recommendation:

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

instead of

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

The same logic should be applied to burnTrigger method.

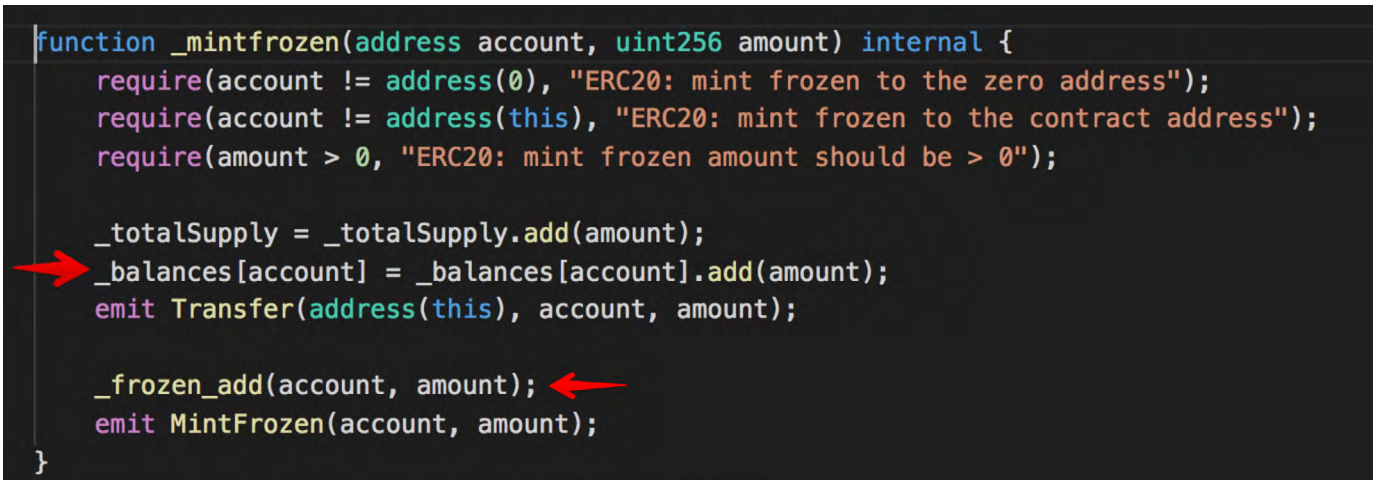
Status:

✓ Fixed

3 Frozen token storage issue

Severity: **MINOR**

In the current implementation, `_frozen_balanceOf` shows how many tokens of the **total** user balance **are frozen**, which requires to change `_balances` mapping in all of 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);
}
```

Recommendation:

To avoid multiple recalculations consider changing the architecture to separately store the number of frozen and active tokens and calculate the `balanceOf` by adding the two values i.e.

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

and `accountBalance` will no longer be necessary.

Status:

✓ Fixed

4 Lack of the Transfer event for the _mintfrozen method

Severity: **MINOR**

The **_mintfrozen** method emits Transfer(address(this), account, amount); event, which is a byproduct of the minting process. However, it does not emit Transfer(account, address(this), amount); event to indicate that freezing took place (like in freezeTokens method).

Recommendation:

Consider adding the event.

Status:

Acknowledged Left as is.

5 Redundant require check

Severity: **MINOR**

The **require**:

```
require(_frozen_balanceOf(from) >= amount, “ERC20: transfer frozen tokens: balance of sender should be >= amount”);
```

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

Recommendation:

Consider removing the check.

Status:

✓ Fixed

6 The approveAndCall method does not validate spender contract response

Severity: **MINOR**

The second version of the token contract has **ApproveAndCall**, but it does not validate the boolean value that spender contract returns.

Recommendation:

It's advisable to have explicit return as boolean.

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

In addition, capitalize **T**okenRecipient interface name.

Status:

✓ Fixed

7 The accountBalance and _frozen_balanceOf methods obfuscate useful information

Severity: **MINOR**

Users should be able to see **the frozen amount of their tokens**. Otherwise, they are unable to properly consider the amount of active tokens.

Recommendation:

Consider making the method public

Status:

Acknowledged

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

8 Unused functionality

Severity: **MINOR**

These 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      }
```

Recommendation:

Consider removing these methods.

Status:

✓ Fixed

Conclusion

During the audit, the experts have discovered 1 **Critical,** 2 **Major** along with 12 other security issues that fall into the Medium and Minor severity groups.

To properly mitigate the discovered security flaws and for the better protection of the token, Peppersec has offered a range of recommendations that, once applied, will stregthen the overall architecture. Some of these security measures has already been implemented by the KICKICO security team, therefore, boosting the resilience of the token.

Appendix 1. Terminology

1 Severity

Assessment 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 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 nature but do not represent any actual bugs or security problems. These issues should be addressed unless there is a clear 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 or 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