verichains

SECURITY AUDIT OF

KARAT TOKEN

Public Report

Jul 24, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Jul 24, 2024. We would like to thank the Karat for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Karat Token. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the source code, along with some recommendations. Karat team has acknowledged and resolved some of these issues.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Karat Token

Unleash KARAT and shape your destiny in Karat Galaxy! Dive into thrilling cosmic adventures with multiple games, mining, NFTs, and DeFi. Explore a blockchain-powered universe like never before where every move impacts your journey.

Target enemy ships, fire your turret, and collect precious gold coins. Use strategic attacks and powerful weapons to dominate the cosmos. Optimize your setups for maximum rewards and become the ultimate Space Pirate!

Uncover KARAT, the pulsating core of Karat Galaxy. With only 100 million tokens ever to be created, each KARAT is a rare and invaluable asset in the universe of Karat Galaxy. Engage in thrilling adventures and forge your path among the stars with this limited and powerful digital currency.

Immerse yourself in the powerhouse of decentralized finance (DeFi) within Karat Galaxy. Utilize KARAT tokens to stake, earn rewards, and thrive in a vibrant in-game economy. Karat Galaxy' DeFi ecosystem ensures transparency, fairness, and security, empowering every player in their cosmic journey.

1.2. Audit Scope

This audit focused on identifying security flaws in code and the design of the Karat Token. It was conducted on commit `a31b41d8066c0eefa942721adb54e5041dc9f014` from git repository `https://github.com/fcwrsmall/solidity.karat`.

1.3. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence

- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Karat acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Karat understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Karat agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Karat will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Karat, the final report will be considered fully accepted by the Karat without the signature.


verichains

2. AUDIT RESULT

2.1. Overview

The Karat Token was written in `Solidity` language, with the required version to be in range `>=0.8.0 <0.9.0`.

The contract is a custom ERC20 token with fixed-capacity, burnable, and fee-on-transfer features. The `balanceOf` function is customized to work with the Uniswap V3 liquidity pool. Most of the contract's configuration properties can be set by the owner, such as the LP address, buy/sell fee tax ratios, buy/sell enable flags, etc.

The total supply of the token is fixed at 100 million KARAT tokens, which are minted to the owner's address upon deployment. The table below describes some properties of the audited Karat Token (as of the report writing time).

PROPERTY	VALUE
Name	KARAT
Symbol	KARAT
Decimals	18
Total Supply	$100{,}000{,}000 \times 10^{18}$ (it represents 100 million tokens)

Table 2. The Karat Token properties

2.2. Findings

During the audit process, the audit team found some vulnerabilities and recommendations in the given version of Karat Token.

#	Issue	Severity	Status
1	Fee-on-transfer token should not be used with Uniswap V3	MEDIUM	Acknowledged
2	Unnecessary complex logic when configuring parameters	INFORMATIVE	Acknowledged

2.2.1. Fee-on-transfer token should not be used with Uniswap V3 MEDIUM

Affected files:

- contracts/fi-panlaxy/karat.sol

The Karat token is a fee-on-transfer token that charges a tax fee on every transfer, which is not compatible with Uniswap V3. Modifying the `balanceOf` function to bypass the tax fee check is not recommended, as this can cause the token to become incompatible with the current DeFi ecosystem (the `block.number` should not affect the balance).

For example, after tokens are sent from the LP to the `recipient`, any other protocol that depends on the `balanceOf(recipient)` in the same block will get an incorrect result.

```
function balanceOf(address account) override external view virtual returns(uint256) {
    unchecked {
        Account memory a = _accounts[account];      /// a workaround for some V3 checking
        return (uint48(block.number) == a.blockNo) ? a.blockView : a.balance; // AUDIT: NOT
RECOMMENDED
    }
}

function _beforeTransfer(uint256 balance, address from, address to, uint256 amount)
internal virtual returns(uint256) {
    unchecked {
        require(balance >= amount,'$');
        if(from != msg.sender) {                    /// non-owner caller must check
allowance
            uint256 allowed = _allowances[from][msg.sender];
            if(allowed < MAX256) {
                require(allowed >= amount,'%');
                _allowances[from][msg.sender] = allowed-amount;
            }
        }
        if(to == address(this))                     /// tokens sent to this contract will
be treated as donation to 'fi'
            return amount;
        address fi = _fi;
        if((to == fi)||(from == fi))                /// no restriction for 'fi' contract
            return amount;
        Defi memory defi = _defi;
        uint256 tax = (from == defi.lp) ? _getCommission(defi.enables,defi.ppmBuy,amount) :
                      (to == defi.lp) ?
_getCommission(defi.enables>>1,defi.ppmSell,amount) :
                                        0;
        if(tax == 0)
            return amount;
        _afterTransfer(from,address(this),tax,tax);
        return amount-tax;
    }
}
```

RECOMMENDATION

The tax fee should be collected from the LP pool instead of the transfer function. This will make the token compatible with all other DeFi protocols, and less likely to cause issues in the future.

By the way, customizing the transfer logic by splitting it into multiple functions like `_beforeTransfer` and `_afterTransfer` in this case is quite complex and error-prone. Specifically, calling `_afterTransfer` for tax inside `_beforeTransfer` does not make any sense and dramatically reduces the readability of the code. It is recommended to simplify or refactor the transfer logic to avoid potential issues.

UPDATES

- **Jul 24, 2024**: This issue has been acknowledged by the Karat team.

2.2.2. Unnecessary complex logic when configuring parameters INFORMATIVE

Affected files:

- contracts/fi-panlaxy/karat.sol

The implementation for configuring parameters in the Karat token contract is unnecessarily complex. The `_config` function is used to set different parameters based on the `op` parameter, which can be simplified by using separate functions for each configuration parameter.

The ownership check logic should be separated using a modifier, which will improve the readability of the code.

Overuse of bitwise operators in `_enable` function just make the function harder to read and understand. It is recommended to use a more straightforward approach to set the enable flags.

```
function _enable(uint8 b, bool enable) internal virtual {
    unchecked {
        if(enable) _defi.enables |= b;
        else _defi.enables &= ~b;
    }
}

function _config(uint256 op, address to, uint256 n) internal virtual returns(bool) {
    unchecked {
        require(msg.sender == _owner,"Denied!");
        if(op == transferOwner_ ) _owner = to; else /// transfer owner, to=0 to renouce
ownership
        if(op == fi_            ) _fi = to; else
        if(op == lp_            ) _defi.lp = to; else
        if(op == buy_           ) _enable(0x1,n != 0); else
        if(op == sell_          ) _enable(0x2,n != 0); else
        if(op == ppmbuy_        ) _defi.ppmBuy = uint16(n); else
        if(op == ppmsell_       ) _defi.ppmSell = uint16(n); else
```

```
            return false;
        return true;
    }
}

function transferOwner(address to) override external virtual {
    if(_config(transferOwner_,to,0))
        emit TransferOwner(msg.sender,to);
}
//////////////////////////////////////////////////////
function configV3lp(address lp) override external virtual {
    _config(lp_,lp,0);
}
//////////////////////////////////////////////////////
function configV3Buy(uint256 ppm) override external virtual {
    _config(ppmbuy_,_0,ppm);
}
//////////////////////////////////////////////////////
function configV3Sell(uint256 ppm) override external virtual {
    _config(ppmsell_,_0,ppm);
}
//////////////////////////////////////////////////////
function enableBuy(bool enable) override external virtual {
    _config(buy_,_0,enable ? 1 : 0);
}
//////////////////////////////////////////////////////
function enableSell(bool enable) override external virtual {
    _config(sell_,_0,enable ? 1 : 0);
}
```

UPDATES

- **Jul 24, 2024**: This issue has been acknowledged by the Karat team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Jul 24, 2024*	Public Report	Verichains Lab

Table 3. Report versions history