KNOWNSEC

Blockchain Lab

Smart Contract Audit Report

Version description

The revision	Date	Revised	Version
Write documentation	20211119	**KNOWNSEC Blockchain Lab**	V1.0

Document information

Title	Version	Document Number	Type
KAKA Smart Contract Audit Report	V1.0	**28d304a500fa411599200ddfc4ee9cda**	**Open to project team**

Statement

KNOWNSEC Blockchain Lab only issues this report for facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities for this. KNOWNSEC Blockchain Lab is unable to determine the security status of its smart contracts and is not responsible for the facts that will occur or exist in the future. The security audit analysis and other content made in this report are only based on the documents and information provided to us by the information provider as of the time this report is issued. KNOWNSEC Blockchain Lab 's assumption: There is no missing, tampered, deleted or concealed information. If the information provided is missing, tampered with, deleted, concealed or reflected in the actual situation, KNOWNSEC Blockchain Lab shall not be liable for any losses and adverse effects caused thereby.

Directory

1. Summarize

The effective test time of this report is from **November 15, 2021 to November 19, 2021.** During this period, the security and standardization of the **token code of the KAKA smart contract** will be audited and used as the statistical basis for the report.

The scope of this smart contract security audit does not include external contract calls, new attack methods that may appear in the future, and code after contract upgrades or tampering. (With the development of the project, the smart contract may add a new pool , New functional modules, new external contract calls, etc.), does not include front-end security and server security.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 6). **The smart contract code of the KAKA** is comprehensively assessed as **PASS**.

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

KNOWNSEC Attest information:

classification	information
report number	28d304a500fa411599200ddfc4ee9cda
report query link	https://attest.im/attestation/searchResult?qurey=28d304a500fa411599200ddfc4ee9cda

2. Item information

2.1. Item description

KAKA NFT WORLD is a competitive gaming ecosystem platform focusing on the metaverse blockchain gaming sector. It is committed to combining the application of NFT+DEFI in the ecosystem, constructing a cross-chain bridge based on the concept of global decentralization, integrating various IPs of global brands, and creating Decentralized Autonomous Management (DAO). It aims to create a prediction agreement perpendicular to the e-sports prediction market, and an open, transparent, decentralized, and complete project for the ecosystem. The main sectors are divided into the two following categories: providing services such as R&D, sales, trading, circulation, e-sports, and trendy games for game and art NFTs; focusing on the prediction market in the field of e-sports (Prediction Market).

2.2. The project's website

https//www.kakanft.com

2.3. White Paper

https://docs.google.com/presentation/d/1m-ictcWijAi-WxxuP2MpfLJn-tU8lltw/edit#slide=id.p1

2.4. Review version code

https://bscscan.com/address/0x3cfa0da2A460D938C91bc5df333e729F14874b7A#contracts

2.5. Contract file and Hash/contract deployment address

The contract documents	MD5
Ownable.sol	D219C978070026BB046951EC2BC2C8AB

ERC20.sol	739BD2CA1487F5B8797081F38A7AA08F
IERC20.sol	A511A1C838E3382005DE7F88D1F97A88
IERC20Metadata.sol	756AA7ADFBCBC6061B38E277733A541D
SafeERC20.sol	D2176CDBFA3C62AF113C006795420103
Address.sol	4C9AE295CCCA911F91532B69FF395BC1
Context.sol	D57261F2C7858423B939641C580F57D3
KAKA_Token.sol	B93883E9E949C7D55E87C666D4435914

3. External visibility analysis

3.1. KAKA_Token contracts

KAKA_Token					
funcName	**visibility**	**state changes**	**decorator**	**payable reception**	**instructions**
setWhiteList	public	True	onlyOwner	----	---
setWhiteListStatus	public	True	onlyOwner	---	---
transfer	public	True	---	---	---
transferFrom	public	True	----	---	---
burn	public	---	--	---	---
burnFrom	public	----	---	---	---

4. Code vulnerability analysis

4.1. Summary description of the audit results

Audit results			
audit project	audit content	condition	description
Business security detection	Ownable contract permission abandonment, transfer, reset	Pass	After testing, there is no security issue.
	KAKA_Token contract setting whitelist	Pass	After testing, there is no security issue.
	KAKA_Token contract token transfer function	Pass	After testing, there is no security issue.
	KAKA_Token contract burning function	Pass	After testing, there is no security issue.
Code basic vulnerability detection	Compiler version security	Pass	After testing, there is no security issue.
	Redundant code	Pass	After testing, there is no security issue.
	Use of safe arithmetic library	Pass	After testing, there is no security issue.
	Not recommended encoding	Pass	After testing, there is no security issue.
	Reasonable use of require/assert	Pass	After testing, there is no security issue.
	fallback function safety	Pass	After testing, there is no security issue.

	tx.origin authentication	Pass	After testing, there is no security issue.
	Owner permission control	Pass	After testing, there is no security issue.
	Gas consumption detection	Pass	After testing, there is no security issue.
	call injection attack	Pass	After testing, there is no security issue.
	Low-level function safety	Pass	After testing, there is no security issue.
	Vulnerability of additional token issuance	Pass	After testing, there is no security issue.
	Access control defect detection	Pass	After testing, there is no security issue.
	Numerical overflow detection	Pass	After testing, there is no security issue.
	Arithmetic accuracy error	Pass	After testing, there is no security issue.
	Wrong use of random number detection	Pass	After testing, there is no security issue.
	Unsafe interface use	Pass	After testing, there is no security issue.
	Variable coverage	Pass	After testing, there is no security issue.
	Uninitialized storage pointer	Pass	After testing, there is no security issue.
	Return value call verification	Pass	After testing, there is no security issue.
	Transaction order dependency detection	Pass	After testing, there is no security issue.
	Timestamp dependent attack	Pass	After testing, there is no security issue.

	Denial of service attack detection	Pass	After testing, there is no security issue.
	Fake recharge vulnerability detection	Pass	After testing, there is no security issue.
	Reentry attack detection	Pass	After testing, there is no security issue.
	Replay attack detection	Pass	After testing, there is no security issue.
	Rearrangement attack detection	Pass	After testing, there is no security issue.

5. Business security detection

5.1. Ownable contract permission abandonment, transfer, reset【Pass】

Audit analysis: Audit the renounceOwnership/transferOwnership/_setOwner functions in the contract. In the Ownable contract, renounceOwnership is used to make the Owner's address empty to achieve the role of permission abandonment; use transferOwnership to transfer permissions; use _setOwner to reset Owner. Set up the new owner. After auditing, the logical design is reasonable and no security issues have been found.

```
function renounceOwnership() public virtual onlyOwner {
    _setOwner(address(0));
} //knownsec //Permission dump


function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address"); // knownsec // Check if the new address is empty
    _setOwner(newOwner);
} // knownsec // Permission transfer


function _setOwner(address newOwner) private {
    address oldOwner = _owner;
    _owner = newOwner;
```

```
        emit OwnershipTransferred(oldOwner, newOwner);// knownsec//Trigger permission transfer event

    } //knownsec // Set new owner
```

Security advice: None.

5.2. KAKA_Token contract setting whitelist【Pass】

Audit analysis: Audit the setWhiteList/setWhiteListStatus functions in the contract. Two methods are used in the KAKA_Token contract to set the black and white list of account addresses. After auditing, the logical design is reasonable and no security issues have been found.

```
    function setWhiteList(address permit, bool b) public onlyOwner returns (bool) {

        whiteList[permit] = b;

        return true;

    }// knownsec//Set whitelist


    function setWhiteListStatus(bool b) public onlyOwner returns (bool) {

        whiteListStatus = b;

        return true;

    }// knownsec//Set whitelist status
```

Security advice: None.

5.3. KAKA_Token contract token transfer function【Pass】

Audit analysis: Audit the transfer/transferFrom function in the contract. The transfer method in the KAKA_Token contract realizes the transfer function of tokens, and uses transferFrom to realize the transfer function of authorized accounts; it also uses the whitelist function to determine whether the account is valid. After auditing, the logical design is reasonable and no security issues have been found.

```
function transfer(address recipient, uint256 amount) public override returns (bool) {
        if (whiteListStatus) {//knownsec//Determine whether the transfer user is legal
            require(!_msgSender().isContract() || whiteList[_msgSender()]);
            require(!recipient.isContract() || whiteList[recipient]);
        }
        _transfer(_msgSender(), recipient, amount);
        return true;
    }//knownsec//transfer


    function transferFrom(address sender, address recipient, uint256 amount) public override returns (bool) {
        if (whiteListStatus) {//knownsec//Determine whether the authorized account and the transfer user are legal
            require(!_msgSender().isContract() || whiteList[_msgSender()]);
            require(!sender.isContract() || whiteList[sender]);
            require(!recipient.isContract() || whiteList[recipient]);
```

```
        }
        _transfer(sender, recipient, amount);
        uint256 currentAllowance = allowance(sender, _msgSender());
        require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");//knownsec//Determine whether the authorized amount is greater than the transfer amount
        _approve(sender, _msgSender(), currentAllowance - amount);// knownsec//Reset authorization limit
        return true;
    }//knownsec//Authorize account transfer
```

Security advice: None.

5.4. KAKA_Token contract burning function【Pass】

Audit analysis: Audit the transfer/transferFrom function in the contract. The transfer method in the KAKA_Token contract realizes the transfer function of tokens, and uses transferFrom to realize the transfer function of authorized accounts; it also uses the whitelist function to determine whether the account is valid. After auditing, the logical design is reasonable and no security issues have been found.

```
function burn(uint256 amount) public returns (bool) {
    _burn(_msgSender(), amount);// knownsec//To burn coins
    require(totalSupply() >= 21000000 ether, "Burn exceeds limit");//knownsec// Determine whether the total number of tokens is greater than 21,000,000
    return true;
}//knownsec//Burn coins
```

```
function burnFrom(address account, uint256 amount) public returns (bool) {
    uint256 currentAllowance = allowance(account, _msgSender());
    require(currentAllowance >= amount, "ERC20: burn amount exceeds allowance");//knownsec//Determine whether the authorized amount is greater than the transfer amount
    _approve(account, _msgSender(), currentAllowance - amount);
    _burn(account, amount);// knownsec//To burn coins
    require(totalSupply() >= 21000000 ether, "Burn exceeds limit");//knownsec// Determine whether the total number of tokens is greater than 21,000,000
    return true;
}//knownsec//Burn authorized tokens
```

Security advice: None.

6. Code basic vulnerability detection

6.1. Compiler version security【Pass】

Check to see if a secure compiler version is used in the contract code implementation.

Detection results: After detection, the smart contract code has developed a compiler version of 0.8.0 or more, there is no security issue.

Security advice: None.

6.2. Redundant code【Pass】

Check that the contract code implementation contains redundant code.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.3. Use of safe arithmetic library【Pass】

Check to see if the SafeMath security abacus library is used in the contract code implementation.

Detection results: The SafeMath security abacus library has been detected in the smart contract code and there is no such security issue.

Security advice: None.

6.4. Not recommended encoding【Pass】

Check the contract code implementation for officially uns recommended or deprecated coding methods.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.5. Reasonable use of require/assert【Pass】

Check the reasonableness of the use of require and assert statements in contract code implementations.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.6. Fallback function safety【Pass】

Check that the fallback function is used correctly in the contract code implementation.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.7. tx.origin authentication【Pass】

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts makes contracts vulnerable to phishing-like attacks.z

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.8. Owner permission control【Pass】

Check that theowner in the contract code implementation has excessive permissions. For example, modify other account balances at will, and so on.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.9. Gas consumption detection【Pass】

Check that the consumption of gas exceeds the maximum block limit.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.10. call injection attack【Pass】

When a call function is called, strict permission control should be exercised, or the function called by call calls should be written directly to call calls.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.11. Low-level function safety【Pass】

Check the contract code implementation for security vulnerabilities in the use of call/delegatecall

The execution context of the call function is in the contract being called, while the execution context of the delegatecall function is in the contract in which the function is currently called.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.12. Vulnerability of additional token issuance【Pass】

Check to see if there are functions in the token contract that might increase the total token volume after the token total is initialized.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.13. Access control defect detection【Pass】

Different functions in the contract should set reasonable permissions, check whether the functions in the contract correctly use pubic, private and other keywords for visibility modification, check whether the contract is properly defined and use modifier access restrictions on key functions, to avoid problems caused by overstepping the authority.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.14. Numerical overflow detection【Pass】

The arithmetic problem in smart contracts is the integer overflow and integer overflow, with Solidity able to handle up to 256 digits (2^256-1), and a maximum number increase of 1 will overflow to get 0. Similarly, when the number is an unsigned type, 0 minus 1 overflows to get the maximum numeric value.

Integer overflows and underflows are not a new type of vulnerability, but they are particularly dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the likelihood is not anticipated, which can affect the reliability and safety of the program.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.15. Arithmetic accuracy error【Pass】

Solidity has a data structure design similar to that of a normal programming language, such as variables, constants, arrays, functions, structures, and so on, and there is a big difference between Solidity and a normal programming language - Solidity does not have floating-point patterns, and all of Solidity's numerical operations result in integers, without the occurrence of decimals, and without allowing the definition of decimal type data. Numerical operations in contracts are essential, and numerical operations are designed to cause relative errors, such as sibling operations: 5/2 x 10 x 20, and 5 x 10/2 x 25, resulting in errors, which can be greater and more obvious when the data is larger.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.16. Incorrect use of random numbers【Pass】

Random numbers may be required in smart contracts, and while the functions and variables provided by Solidity can access significantly unpredictable values, such as block.number and block.timestamp, they are usually either more public than they seem, or are influenced by miners, i.e. these random numbers are somewhat predictable, so malicious users can often copy it and rely on its unpredictability to attack the feature.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.17. Unsafe interface usage【Pass】

Check the contract code implementation for unsafe external interfaces, which can be controlled, which can cause the execution environment to be switched and control contract execution arbitrary code.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.18. Variable coverage【Pass】

Check the contract code implementation for security issues caused by variable overrides.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.19. Uninitialized storage pointer【Pass】

A special data structure is allowed in solidity as a strut structure, while local variables within the function are stored by default using stage or memory.

The existence of store (memory) and memory (memory) is two different concepts, solidity allows pointers to point to an uninitialized reference, while uninitialized local stage causes variables to point to other stored variables, resulting in variable overrides, and even more serious consequences, and should avoid initializing the task variable in the function during development.

Detection results: After detection, the smart contract code does not have the problem.

Security advice: None.

6.20. Return value call verification【Pass】

This issue occurs mostly in smart contracts related to currency transfers, so it is also known as silent failed sending or unchecked sending.

In Solidity, there are transfer methods such as transfer(), send(), call.value(), which can be used to send tokens to an address, the difference being: transfer send failure will be throw, and state rollback; Call.value returns false when it fails to send, and passing all available gas calls (which can be restricted by incoming gas_value parameters) does not effectively prevent reentration attacks.

If the return values of the send and call.value transfer functions above are not checked in the code, the contract continues to execute the subsequent code, possibly with unexpected results due to token delivery failures.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.21. Transaction order dependency【Pass】

Because miners always get gas fees through code that represents an externally owned address (EOA), users can specify higher fees to trade faster. Since blockchain is public, everyone can see the contents of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transactions at a higher cost to preempt the original solution.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.22. Timestamp dependency attack【Pass】

Block timestamps typically use miners' local time, which can fluctuate over a range of about 900 seconds, and when other nodes accept a new chunk, they only need to verify that the timestamp is later than the previous chunk and has a local time error of less than 900 seconds. A miner can profit from setting the timestamp of a block to meet as much of his condition as possible.

Check the contract code implementation for key timestamp-dependent features.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.23. Denial of service attack【Pass】

Smart contracts that are subject to this type of attack may never return to normal operation. There can be many reasons for smart contract denial of service, including malicious behavior as a transaction receiver, the exhaustion of gas caused by the artificial addition of the gas required for computing functionality, the misuse of access control to access the private component of smart contracts, the exploitation of confusion and negligence, and so on.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.24. Fake recharge vulnerability【Pass】

The transfer function of the token contract checks the balance of the transfer initiator (msg.sender) in the if way, when the balances < value enters the else logic part and return false, and ultimately does not throw an exception, we think that only if/else is a gentle way of judging in a sensitive function scenario such as transfer is a less rigorous way of coding.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.25. Reentry attack detection 【Pass】

The call.value() function in Solidity consumes all the gas it receives when it is used to send tokens, and there is a risk of re-entry attacks when the call to the call tokens occurs before the balance of the sender's account is actually reduced.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.26. Replay attack detection 【Pass】

If the requirements of delegate management are involved in the contract, attention should be paid to the non-reusability of validation to avoid replay attacks

In the asset management system, there are often cases of entrustment management, the principal will be the assets to the trustee management, the principal to pay a certain fee to the trustee. This business scenario is also common in smart contracts.

Detection results: The security issue is not present in the smart contract code after detection.

Security advice: None.

6.27. Rearrangement attack detection 【Pass】

A reflow attack is an attempt by a miner or other party to "compete" with a smart contract participant by inserting their information into a list or mapping, giving an attacker the opportunity to store their information in a contract.

Detection results: After detection, there are no related vulnerabilities in the smart contract code.

Security advice: None.

7. Appendix A: Security Assessment of Contract Fund Management

Contract fund management		
The type of asset in the contract	**The function is involved**	**Security risks**
User mortgage token assets	---	**SAFE**
Users mortgage platform currency assets	---	**SAFE**

Check the security of the management of **digital currency assets** transferred by users in the business logic of the contract. Observe whether there are security risks that may cause the loss of customer funds, such as **incorrect recording, incorrect transfer, and backdoor** withdrawal of the **digital currency assets** transferred into the contract.

KNOWNSEC
Blockchain Lab

Official Website
www.knownseclab.com

E-mail
blockchain@knownsec.com

WeChat Official Account