KOK Token Smart Contract Audit

1. Introduction

iosiro was commissioned by the KOK Foundation to conduct an audit on their ERC-20 token smart contracts. The initial audit was performed on 21 February 2020 and a review was performed on 24 February 2020 of a fix implemented to address a high level risk.

This report is organized into the following sections.

- **Section 2 - Executive Summary:** A high-level description of the findings of the audit.
- **Section 3 - Audit Details:** A description of the scope and methodology of the audit.
- **Section 4 - Design Specification:** An outline of the intended functionality of the smart contracts.

- **Section 5 - Detailed Findings:** Detailed descriptions of the findings of the audit.

The information in this report should be used to understand the risk exposure of the smart contracts, and as a guide to improving the security posture of the smart contracts by remediating the issues that were identified. The results of this audit are only a reflection of the source code reviewed at the time of the audit and of the source code that was determined to be in-scope.

2. Executive Summary

This report presents the findings of a smart contract audit performed by iosiro on the KOK Token smart contracts. The purpose of this audit was to achieve the following.

- Ensure that the smart contracts functioned as intended.
- Identify potential security flaws.

Assessing the economics, game theory, or underlying business model of the platform were strictly beyond the scope of this audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered very high risk with regards to cyber attacks. As such, the highest level of security should be observed when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. There are a number of techniques that can help to achieve this, some of which are described below.

- Security should be integrated into the development lifecycle.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed when possible.

At the conclusion of the audit, several informational issues were present. The informational issues included best practice recommendations to improve the functionality of the codebase.

It should be noted that the code quality was relatively poor, as it was not formatted, did not make use of comments, and did not conform to industry standards. It is recommended that the token be reimplemented using the current OpenZeppelin smart contracts to better adhere to industry standards.

3. Audit Details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files is considered to be out-of-scope. Out-of-scope code that interacts with in-scope code is assumed to function as intended and introduce no functional or security vulnerabilities for the purposes of this audit.

3.1.1 Synthetix Smart Contracts

Project Name: ERC-20

Files: 55a501f

3.2 Methodology

A variety of techniques were used while conducting the audit. These techniques are briefly described below.

3.2.1 Code Review

The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high risk areas of the system.

3.2.2 Dynamic Analysis

The contracts were compiled, deployed, and tested in a Ganache test environment. Manual analysis was used to confirm that the code operated at a functional level, and

to verify the exploitability of any potential security issues identified.

3.2.3 Automated Analysis

Tools were used to automatically detect the presence of several types of security vulnerabilities, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. The static analysis results were manually analysed to remove false positive results. True positive results would be indicated in this report. Static analysis was conducted using Slither, Securify, as well as MythX. Tools such as the Remix IDE, compilation output, and linters were also used to identify potential areas of concern.

3.3 Risk Ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High Risk** - The issue could result in a loss of funds for the contract owner or system users.
- **Medium Risk** - The issue resulted in the code specification being implemented incorrectly.
- **Low Risk** - A best practice or design issue that could affect the security of the contract.
- **Informational** - A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed** - The issue was identified during the audit and has since been addressed to a satisfactory level to remove the risk that it posed.

4. Design Specification

The following section outlines the intended functionality of the system at a high level.

4.1 KOK Token Smart Contract

The KOK Token contract is described below.

ERC-20 Token

The token should implement the ERC-20 standard with the following values.

Field	Value
Symbol	KOK
Name	KOK Coin
Decimals	18
Total Supply	5 billion

Ownable

The token should have an owner address, who is able to perform special operations. It should be possible for the owner to transfer their ownership to another address.

Stoppable

It should be possible for the owner to stop and start all transfers in the system.

Blacklist

It should be possible for the owner to add and remove addresses from a blacklist. When an address is blacklisted, it should not be possible for the address to send or receive tokens.

Airdrop

There should be functionality built into the token that allows the owner to pass in an array of addresses and amounts to transfer tokens to.

5. Detailed Findings

The following section includes in depth descriptions of the findings of the audit.

5.1 High Risk

No high risk issues were present at the conclusion of the audit.

5.2 Medium Risk

No medium risk issues were present at the conclusion of the audit.

5.3 Low Risk

No low risk issues were present at the conclusion of the audit.

5.4 Informational

5.4.1 Design Comments

Actions to improve the functionality and readability of the codebase are outlined below.

Use `bool` for Blacklist State

An `int` type was use for storing whether an address was blacklisted. A more intuitive type for this binary state would be a boolean, using `true` or `false` to indicate whether an address was blacklisted.

Use `require` for Blacklist

The `transfer(...)` and `transferFrom(...)` functions returned `false` if attempting to transfer from or to a blacklisted address. While this is within the ERC-20 standard, a more intuitive implementation would revert through a `require` statement. This would prevent third-parties from potentially assuming that a transfer succeeded if the return value of the function was not validated.

Use `require` for `onlyOwner`

The `onlyOwner` modifier returned `false` instead of reverting. It is recommended that a `require` statement be used to explicitly revert if the function is called by a non-owner address.

No Token Creation Transfer Event

In the ERC-20 standard it specifies that a Transfer event be emitted whenever there is movement of tokens. On contract.sol#L63, the total supply is allocated to the `msg.sender`, however no event is emitted. It is recommended that an event transferring the tokens from 0x0 to the address be added.

Interface Incorrectly Defined

On contact.sol#L25 `ERC20Interface` implies that the code should be an interface, however the `contract` keyword was used. It is recommended that contact.sol#L26 be removed and the keyword changed to `interface`.

No Function Visibility

The `stop(...)` and `start(...)` functions had an implied public visibility. It is recommended that a visibility is explicitly set on these functions by using the `public` keyword to improve the readability of the code.

Inexact Compiler Version

The pragma version was not fixed to a specific version, as it specified `^0.4.21`, which would result in using the highest non-breaking version (highest version below `0.5.0`). According to best practice, where possible, all contracts should use the same compiler version, which should be fixed to a specific version. This helps to ensure that contracts do not accidentally get deployed using an alternative compiler, which may pose the risk of unidentified bugs. An explicit version also helps with code reuse, as users would be able to see the author's intended compiler version. It is recommended that the pragma version is changed to a fixed value, for example `0.4.25`.

5.5 Closed

5.5.1 Incorrectly Named Constructor (High Risk)

contract.sol#L62

Description

A function named `KOKContract(...)` was implemented with a comment above it indicating that it was intended to be the contract constructor. As the contract name was `KOK_Contract(...)`, the function did not operate as a constructor. Given the implementation of the function, the first address to call it would claim all of the tokens.

In Solidity 0.4.22, the `constructor` keyword was added to avoid this issue. In Solidity 0.5.0 using a function name to indicate the constructor was disallowed entirely.

Remedial Action

It is recommended that the constructor be modified to use the `constructor` keyword in order to operate as intended. (more info)

Comment

Fixed in 553f8ad.

ABOUT
CONTACT US
SMART CONTRACT AUDITING
PENETRATION TESTING
AUDIT REPORTS
PRIVACY POLICY
TERMS OF SERVICE