BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for KOHENOOR ERC20 Token Smart Contract

Request Date: 2023-12-06
Completion Date: 2023-12-07
Language: Solidity

Contents

Commission

Audited Project	KOHENOOR ERC20 TOKEN Smart Contract
Smart Contract	0x0835cDd017eA7bC4cC187C6e0f8ea2DBE0feA0Dd
Contract Deployer	0xE8Fa4b57BE78bd3e783c67852e38293BfA9bf396
Contract Owner	0xE8Fa4b57BE78bd3e783c67852e38293BfA9bf396
Blockchain Platform	Polygon Mainnet

Block Solutions was commissioned by KOHENOOR ERC20 TOKEN Smart Contract owners to perform an audit of their main smart contract. The purpose of the audit was to achieve the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

KOHENOOR ERC20 TOKEN Smart Contract Properties

Name	KOHENOOR
Symbol	KEN
Decimals	18
Total Supply	1,000,000 KEN
Transfer	1,373
Holders	1,127 Addresses
Tax Address	0x0000000000000000000000000000000000000000
Smart Contract	0x0835cDd017eA7bC4cC187C6e0f8ea2DBE0feA0Dd
Contract Deployer	0xE8Fa4b57BE78bd3e783c67852e38293BfA9bf396
Contract Owner	0xE8Fa4b57BE78bd3e783c67852e38293BfA9bf396
Blockchain Platform	Polygon Mainnet

Contract Functions

Executables

i. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
ii. function approve(address spender, uint256 amount) public virtual override returns (bool)
iii. function blackList(address addr) external onlyOwner whenNotPaused
iv. function burn(uint256 amount) public override onlyOwner whenNotPaused
v. function burnFrom(address from,uint256 amount) public override onlyOwner whenNotPaused
vi. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
vii. function mint(address to, uint256 amount) external onlyOwner whenNotPaused
viii. function pause() external onlyOwner
ix. function removeFromBlacklist(address addr) external onlyOwner whenNotPaused
x. function renounceOwnership() public override onlyOwner whenNotPaused
xi. function setDeflationConfig(uint256 _deflationBPS) external onlyOwner whenNotPaused
xii. function setDocumentUri(string memory newDocumentUri) external onlyOwner whenNotPaused
xiii. function setMaxTokenAmountPerAddress(uint256 newMaxTokenAmount) external onlyOwner whenNotPaused
xiv. function setTaxConfig(address _taxAddress,uint256 _taxBPS) external onlyOwner whenNotPaused
xv. function transfer(address to,uint256 amount) public virtual override whenNotPaused validateTransfer(msg.sender, to) returns (bool)
xvi. function transferFrom(address from,address to,uint256 amount) public virtual override whenNotPaused validateTransfer(from, to) returns (bool)
xvii. function transferOwnership(address newOwner) public override onlyOwner whenNotPaused
xviii. function unpause() external onlyOwner
xix. function updateWhitelist(address[] calldata updatedAddresses) external onlyOwner whenNotPaused

Checklist

Compiler errors.	Passed
Possible delays in data delivery.	Passed
Timestamp dependence.	Passed
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with block gas limit.	Passed
Methods execution permissions.	Passed
Economy model of the contract.	Passed
Private user data leaks.	Passed
Malicious Events Log.	Passed
Scoping and Declarations.	Passed
Uninitialized storage pointers.	Passed
Arithmetic accuracy.	Passed
Design Logic.	Passed
Impact of the exchange rate.	Passed
Oracle Calls.	Passed
Cross-function race conditions.	Passed
Fallback function security.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed

Whitepaper-Website-Contract correlation.	Not Checked
Front Running.	Not Checked

Contract's Functions

KOHENOOR ERC20 Token Contract

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the authorized address.

```
function transferOwnership(address newOwner) public override onlyOwner
 whenNotPaused {
  super.transferOwnership(newOwner);
}
```

Leaves the contract without owner. It will not be possible to call `onlyOwner` functions anymore. Can only be called by the current owner.

```
function renounceOwnership() public override onlyOwner whenNotPaused {
  super.renounceOwnership();
}
```

Atomically decreases the allowance granted to `spender` by the caller. This is an alternative to {approve} that can be used as a mitigation for problems described in {IERC20-approve}.
Emits an {Approval} event indicating the updated allowance.
Requirements:

- `spender` cannot be the zero address.
- `spender` must have allowance for the caller of at least `subtractedValue`.

```
function decreaseAllowance(address spender, uint256 subtractedValue) public
virtual returns (bool) {
    address owner = _msgSender();
    uint256 currentAllowance = allowance(owner, spender);
    require(currentAllowance >= subtractedValue,
    "ERC20: decreased allowance below zero");
    unchecked {
        _approve(owner, spender, currentAllowance - subtractedValue);
    }
    return true;
}
```

It allows to burn a predefined number of tokens. "amount" is the number of tokens to burn.

- Only callable by the owner
- Only callable if the token is not paused

- Only callable if the token supports burning

```
function burn(uint256 amount) public override onlyOwner whenNotPaused {
  if (!isBurnable()) {
    revert BurningNotEnabled();
  }
  super.burn(amount);
}
```

Atomically increases the allowance granted to `spender` by the caller. This is an alternative to {approve} that can be used as a mitigation for problems described in {IERC20-approve}.
Emits an {Approval} event indicating the updated allowance.
Requirements:
`spender` cannot be the zero address.

```
function increaseAllowance(address spender, uint256 addedValue) public
virtual returns (bool) {
    address owner = _msgSender();
    _approve(owner, spender, allowance(owner, spender) + addedValue);
    return true;
}
```

Owner of this contract can mint token on any address by executing this function.

```
function mint(address to, uint256 amount) external onlyOwner whenNotPaused {
  if (!isMintable()) {
    revert MintingNotEnabled();
  }
  if (isMaxAmountOfTokensSet()) {
    if (balanceOf(to) + amount > maxTokenAmountPerAddress) {
      revert DestBalanceExceedsMaxAllowed(to);
    }
  }
  if (isBlacklistEnabled()) {
    if (_isBlacklisted[to]) {
      revert RecipientBlacklisted(to);
    }
```

Owner of this contract can pause the contract state.

```
function pause() external onlyOwner {
  if (!isPausable()) {
    revert PausingNotEnabled();
  }
  _pause();
}
```

Owner of this contract can remove any address from black list addresses list.

```
function removeFromBlacklist(address addr) external onlyOwner whenNotPaused {
  Helpers.validateAddress(addr);
  if (!isBlacklistEnabled()) {
    revert BlacklistNotEnabled();
  }
  if (!_isBlacklisted[addr]) {
    revert AddrAlreadyUnblacklisted(addr);
  }
  delete _isBlacklisted[addr];
  emit UserUnBlacklisted(addr);
}
```

Owner of this contract can set the deflation tax.

```
function setDeflationConfig(uint256 _deflationBPS) external onlyOwner
whenNotPaused {
  if (!isDeflationary()) {
    revert TokenIsNotDeflationary();
  }
  if (_deflationBPS > MAX_BPS_AMOUNT) {
    revert InvalidDeflationBPS(_deflationBPS);
  }
  deflationBPS = _deflationBPS;
  emit DeflationConfigSet(_deflationBPS);
}
```

Owner of this contract can set the document URI.

```
function setDocumentUri(string memory newDocumentUri) external onlyOwner
whenNotPaused {
  if (!isDocumentUriAllowed()) {
    revert DocumentUriNotAllowed();
  }
  documentUri = newDocumentUri;
  emit DocumentUriSet(newDocumentUri);
}
```

Owner of this contract can set the maximum token amount.

```
function setMaxTokenAmountPerAddress(uint256 newMaxTokenAmount) external
 onlyOwner whenNotPaused {
  if (!isMaxAmountOfTokensSet()) {
    revert MaxTokenAmountNotAllowed();
  }
  if (newMaxTokenAmount <= maxTokenAmountPerAddress) {
    revert MaxTokenAmountPerAddrLtPrevious();
  }
  maxTokenAmountPerAddress = newMaxTokenAmount;
  emit MaxTokenAmountPerSet(newMaxTokenAmount);
}
```

Owner of this contract can set the tax address and tax value.

```
function setTaxConfig(address _taxAddress,uint256 _taxBPS) external onlyOwner
 whenNotPaused {
  if (!isTaxable()) {
    revert TokenIsNotTaxable();
  }
  if (_taxBPS > MAX_BPS_AMOUNT) {
    revert InvalidTaxBPS(_taxBPS);
  }
  Helpers.validateAddress(_taxAddress);
  taxAddress = _taxAddress;
  taxBPS = _taxBPS;
  emit TaxConfigSet(_taxAddress, _taxBPS);
}
```

Owner of this contract set the contract state to un pause.

```
function unpause() external onlyOwner {
  if (!isPausable()) {
    revert PausingNotEnabled();
  }
  _unpause();
}
```

Owner of this contract adds new whitelist addresses

```
function updateWhitelist(address[] calldata updatedAddresses) external
onlyOwner whenNotPaused {
  if (!isWhitelistEnabled()) {
    revert WhitelistNotEnabled();
  }
  _clearWhitelist();
  _addManyToWhitelist(updatedAddresses);
  whitelistedAddresses = updatedAddresses;
  emit UsersWhitelisted(updatedAddresses);
}
```

Destroys a `value` amount of tokens from `account`, deducting from the caller's allowance. See {ERC20-_burn} and {ERC20-allowance}.
Requirements:
the caller must have allowance for ``accounts``'s tokens of at least `value`.

```
function burnFrom(address from,uint256 amount) public override onlyOwner
whenNotPaused {
  if (!isBurnable()) {
    revert BurningNotEnabled();
  }
  super.burnFrom(from, amount);
}
```

Owner of this contract can blacklist any address.

```
function blackList(address addr) external onlyOwner whenNotPaused {
  Helpers.validateAddress(addr);
  if (!isBlacklistEnabled()) {
    revert BlacklistNotEnabled();
  }
  if (_isBlacklisted[addr]) {
    revert AddrAlreadyBlacklisted(addr);
  }
  if (isWhitelistEnabled() && whitelist[addr]) {
    revert CannotBlacklistWhitelistedAddr(addr);
  }
  _isBlacklisted[addr] = true;
  emit UserBlacklisted(addr);
}
```

Approve the passed address to spend the specified number of tokens on behalf of msg. sender. “spender” is the address which will spend the funds. “amount” the number of tokens to be spent.

```
function approve(address spender, uint256 amount) public virtual override
 returns (bool) {
    address owner = _msgSender();
    _approve(owner, spender, amount);
    return true;
}
```

This will transfer token for a specified address “to” is the address to transfer. “amount” is the amount to be transferred.

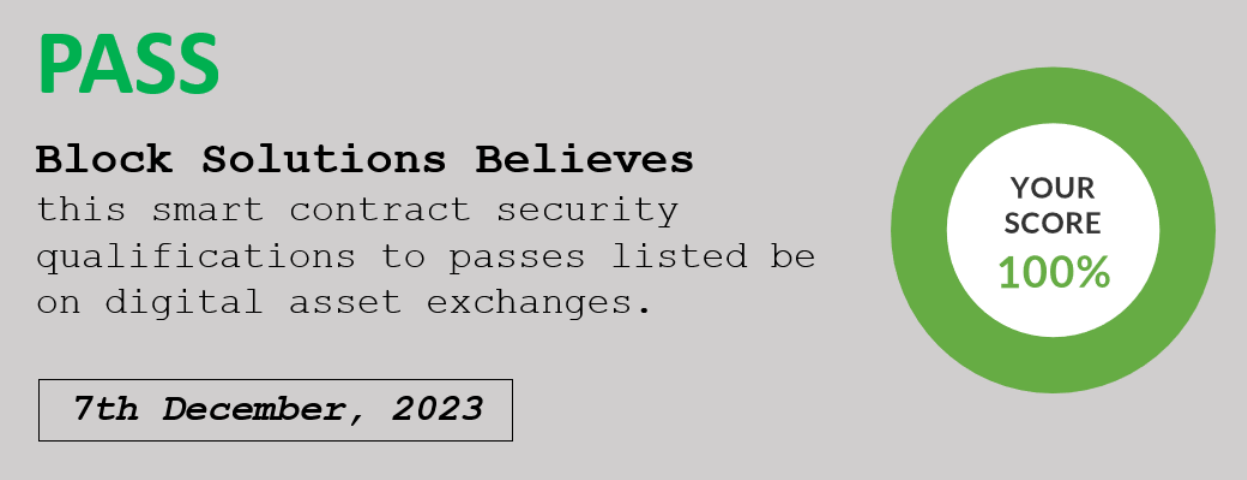
```
function transfer(address to,uint256 amount) public virtual
override whenNotPaused validateTransfer(msg.sender, to) returns (bool)
{
  uint256 taxAmount = _taxAmount(msg.sender, amount);
  uint256 deflationAmount = _deflationAmount(amount);
  uint256 amountToTransfer = amount - taxAmount - deflationAmount;
  if (isMaxAmountOfTokensSet()) {
    if (balanceOf(to) + amountToTransfer > maxTokenAmountPerAddress) {
      revert DestBalanceExceedsMaxAllowed(to);
    }
  }
  if (taxAmount != 0) {
    _transfer(msg.sender, taxAddress, taxAmount);
  }
  if (deflationAmount != 0) {
    _burn(msg.sender, deflationAmount);
  }
  return super.transfer(to, amountToTransfer);
}
```

Transfer tokens from the “from” account to the “to” account. The calling account must already have sufficient tokens approved for spending from the “from” account and “from” account must have sufficient balance to transfer. “to” must have sufficient allowance to transfer.

```
function transferFrom(address from,address to,uint256 amount) public virtual
 override whenNotPaused validateTransfer(from, to) returns (bool)
{
  uint256 taxAmount = _taxAmount(from, amount);
  uint256 deflationAmount = _deflationAmount(amount);
  uint256 amountToTransfer = amount - taxAmount - deflationAmount;
  if (isMaxAmountOfTokensSet()) {
    if (balanceOf(to) + amountToTransfer > maxTokenAmountPerAddress) {
      revert DestBalanceExceedsMaxAllowed(to);
    }
  }

  if (taxAmount != 0) {
    _transfer(from, taxAddress, taxAmount);
  }
```

Testing Summary



‘

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert () misuse	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: Passed

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is **Well-Secured.** Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.

We used various tools like Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Quick Stat section.
We found critical, 0 high, 7 medium and 0 low level issues.

Code Quality

The KOHENOOR ERC20 Token Smart Contract protocol consists of one smart contract. Libraries used in KOHENOOR ERC20 Token Smart Contract are part of its logical algorithm. They are smart contracts which contain reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in protocol. The BLOCKSOLUTIONS team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way. Overall, the code is not commented. Commenting can provide rich documentation for functions, return variables and more.

Documentation

As mentioned above, it's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic. We were given a KOHENOOR ERC20 Token Smart Contract smart contract code in the form of File.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well-known industry standard open-source projects. And even core code blocks are written well and systematically. This smart contract does not interact with other external smart contracts.

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical

No Critical severity vulnerabilities were found.

High

No High severity vulnerabilities were found

Medium

1. **Mint function**
   The contract may contain additional issuance functions, which could maybe generate a large number of tokens, resulting in significant fluctuations in token prices. It is recommended to confirm with the project team whether it complies with the token issuance instructions.
2. **Functions that can suspend trading**
   If a suspend able code is included, the token maybe neither be bought nor sold (honeypot risk).
3. **Blacklist function**
   The blacklist function is included. Some addresses may not be able to trade normally (honeypot risk).

4. **Whitelist function**

   The whitelist function is included. Some addresses may not be able to trade normally (honeypot risk).

5. **PRESENCE OF BURN FUNCTION**

   The tokens can be burned in this contract. Burn functions are used to increase the total value of the tokens by decreasing the total supply.

6. **PAUSABLE CONTRACTS**

   This is a Pausable contract. If a contract is Pausable, it allows privileged users or owners to halt the execution of certain critical functions of the contract in case malicious transactions are found.

7. **OVERPOWERED OWNERS**

   The contracts are using 14 functions that can only be called by the owners. Giving too many privileges to the owners via critical functions might put the user's funds at risk if the owners are compromised or if a rug-pulling attack happens.

Low

No Low severity vulnerabilities were found.

Conclusion

The Smart Contract code passed the audit successfully with some considerations to take. There were 7 medium warnings raised. We were given a contract code. And we have used all possible tests based on given objects as files. So, it is good to go for production. Since possible test cases can be unlimited for such extensive smart contract protocol, hence we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything. Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in Quick Stat section of the report. Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract is "Well Secured".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an

appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally, follow a process of first documenting the suspicion with unresolved questions, then

confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.