Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	RoRa Gold
Website:	https://roracoin.com
Platform:	Ethereum
Language:	Solidity
Date:	June 4th, 2022

Table of contents

Email: audit@EtherAuthority.io

`

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Email: audit@EtherAuthority.io

Introduction

EtherAuthority was contracted by the RoRa team to perform the Security audit of the RoRa Gold (RORAG) smart contract code. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on June 4th, 2022.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

- RoRa Gold (RORAG) is an ERC-20 token with Access Control and ERC-1404 transfer restrictions.
- It has functions like hasRole, grantRole, revokeRole, renounceRole, setTimeLock, upgradeTransferRules, upgradeTransferRules, removeTimeLock, etc.

Audit scope

Name	**Code Review and Security Analysis Report for RoRa Gold Token Smart Contract**
Platform	**Ethereum / Solidity**
File	RORAGold.sol
File MD5 Hash	B9C25B43B7E1F1211F4893EF2F9B79B5
Online Code Link	0x6e27d1ab495686a429a1ae3bc7ccabc8b0530eba
Audit Date	June 4th, 2022

Email: audit@EtherAuthority.io

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: RORA Gold • Symbol: RORAG • Decimals: 0 • Initial Supply: 1,500,000 Tokens • Max Supply: very large amount	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, Customer`s solidity based smart contracts are **“Secured”**. This token contract does contain owner control, which does not make it fully decentralized.

We used various tools like Slither, Solhint 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 Audit overview section. General overview is presented in AS-IS section and all identified issues can be found in the Audit overview section.

We found 0 critical, 1 high, 0 medium and 0 low and some very low level issues. These issues are fixed / acknowledged by the RoRa team.

Investors Advice: Technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical 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	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Passed
Gas Optimization	“Out of Gas” Issue	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: **PASSED**

Code Quality

This audit scope has 1 smart contract. Smart contract contains Libraries, Smart contracts, inherits and Interfaces. This is a compact and well written smart contract.

The libraries in RoRa Gold Token are part of its logical algorithm. A library is a different type of smart contract that contains 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 the RoRa Gold Token.

The RoRa Gold Token team has **not** provided scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are **not** commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a RoRa Gold Token smart contract code in the form of an Etherscan weblink. The hash of that code is mentioned above in the table.

As mentioned above, code parts are not well commented. But the contract is straightforward so it's easy to understand its programming logic.

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.

Apart from libraries, its functions are used in external smart contract calls.

AS-IS overview

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	name	read	Passed	No Issue
3	symbol	read	Passed	No Issue
4	decimals	read	Passed	No Issue
5	totalSupply	read	Passed	No Issue
6	balanceOf	read	Passed	No Issue
7	transfer	write	Passed	No Issue
8	allowance	read	Passed	No Issue
9	approve	write	Passed	No Issue
10	transferFrom	write	Passed	No Issue
11	increaseAllowance	write	Passed	No Issue
12	decreaseAllowance	write	Passed	No Issue
13	_transfer	internal	Passed	No Issue
14	_mint	internal	Passed	No Issue
15	_burn	internal	Passed	No Issue
16	_approve	internal	Passed	No Issue
17	_spendAllowance	internal	Passed	No Issue
18	_beforeTokenTransfer	internal	Passed	No Issue
19	_afterTokenTransfer	internal	Passed	No Issue
20	paused	read	Passed	No Issue
21	whenNotPaused	modifier	Passed	No Issue
22	whenPaused	modifier	Passed	No Issue
23	_pause	internal	Passed	No Issue
24	_unpause	internal	Passed	No Issue
25	onlyRole	modifier	Passed	No Issue
26	supportsInterface	read	Passed	No Issue
27	hasRole	read	Passed	No Issue
28	_checkRole	internal	Passed	No Issue
29	getRoleAdmin	read	Passed	No Issue
30	grantRole	write	Passed	No Issue
31	revokeRole	write	Passed	No Issue
32	renounceRole	write	Passed	No Issue
33	_setupRole	internal	Passed	No Issue
34	_setRoleAdmin	internal	Passed	No Issue
35	_grantRole	internal	Passed	No Issue
36	_revokeRole	internal	Passed	No Issue
37	decimals	read	Passed	No Issue
38	pause	external	access only Role	No Issue
39	unpause	external	access only Role	No Issue
40	mint	external	access only Role	No Issue
41	burn	external	access only Role	Refer audit findings

42	transfer	write	Passed	No Issue
43	transferFrom	write	Passed	No Issue
44	setPermission	external	access only Role	No Issue
45	getPermission	external	Passed	No Issue
46	setTimeLock	external	access only Role	No Issue
47	removeTimeLock	external	access only Role	No Issue
48	getTimeLock	external	Passed	No Issue
49	enforceTransferRestrictions	read	Passed	No Issue
50	detectTransferRestriction	read	Passed	No Issue
51	messageForTransferRestriction	read	Passed	No Issue
52	renounceRole	write	Passed	No Issue
53	revokeRole	write	access only Role	No Issue
54	grantRole	write	access only Role	No Issue
55	upgradeTransferRules	external	access only Role	No Issue

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
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.

Email: audit@EtherAuthority.io

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

(1) Admin can burn anyone's tokens:

```
// The abilty to burn from any address required because of regulatory requirements.
// Ability to burn an address is by board decision only.
// It can only be called by the Mint Admin role which is a protected wallet.
function burn(address from, uint256 amount) external whenNotPaused onlyRole(MINT_ADMIN_ROLE) {
    _burn(from, amount);
}
```

A specific admin role can burn unlimited tokens of any wallet. This creates FUD in the user's mind as they may fear that if the owner's private key is compromised, then their assets also can be burned.

Resolution: We suggest not allowing burning of any user's tokens to any user, not even the owner. On another hand, the user can be able to burn his own tokens if he wishes.

Status: This issue is acknowledged by the RoRa team. This is their response, "Only the Minter role can burn anyones token: We do this for regulatory purposes. To try to limit this we designed the contract that only the Mint Role can perform this and this Role must be separate for the Contract Admin Role. The concept is that the Mint Admin Role is a special wallet that is secured so the board must authorize its use."

Medium

No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Very Low / Informational / Best practices:

(1) Multiple Pragma:

```
pragma solidity ^0.8.0;

/**
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        return msg.data;
    }
}

// File: @openzeppelin/contracts/security/Pausable.sol


// OpenZeppelin Contracts v4.4.1 (security/Pausable.sol)

pragma solidity ^0.8.0;
```

There are multiple pragma added to code with different solidity versions

Resolution: We suggest keeping only one pragma on top of the contract code.

Status: Acknowledged

(2) Tokens can be minted a very large amount

```
if (maxTotalSupply_ == 0) maxTotalSupply_ = type(uint256).max;
maxTotalSupply = maxTotalSupply_;
```

This really sets max token supply to be a very large number, which means the owner can mint a very large number of tokens.

Status: Acknowledged

Centralization

This smart contract has some functions which can be executed by the Admin (Owner) only. If the admin wallet private key would be compromised, then it would create trouble. Following are Admin functions:

- pause: Owner can trigger stopped state.
- unpause: Owner can return to normal state.
- mint: Owner can mint amount.
- burn: Owner can burn the amount from the address.
- setPermission: Owner can set permissions.
- setTimeLock: Owner can set timestamp.
- removeTimeLock: Owner can remove timestamp.
- revokeRole: Owner can revoke role address.
- grantRole: Owner can grant role address.
- upgradeTransferRules: Owner can upgrade transfer rules.

To make the smart contract 100% decentralized, we suggest renouncing ownership in the smart contract once its function is completed.

Conclusion

We were given a contract code. And we have used all possible tests based on given objects as files. We have observed one major issue, but that is acknowledged by the RoRa team as a necessary feature. So, **it's good to go for the production**.

Since possible test cases can be unlimited for such smart contracts protocol, 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 the As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed smart contract, based on standard audit procedure scope, is **"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.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

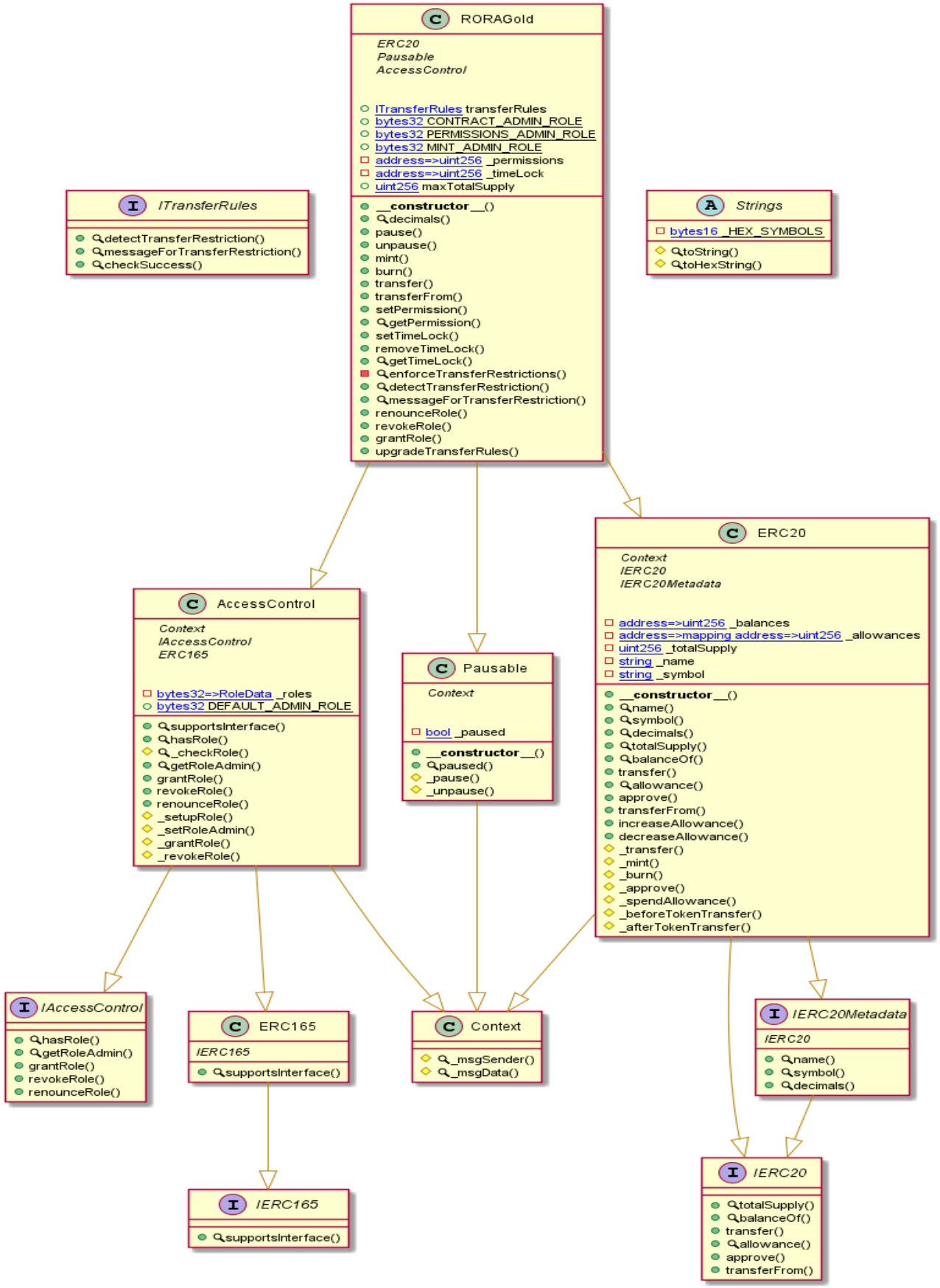
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can’t guarantee explicit security of the audited smart contracts.

Appendix

Code Flow Diagram - RoRa Gold Token



Slither Results Log

Slither Log >> RORAGold.sol

```
INFO:Detectors:
AccessControl._setupRole(bytes32,address) (RORAGold.sol#201-203) is never used and should be removed
Context._msgData() (RORAGold.sol#104-106) is never used and should be removed
Strings.toHexString(uint256) (RORAGold.sol#55-66) is never used and should be removed
Strings.toString(uint256) (RORAGold.sol#35-53) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Pragma version^0.8.0 (RORAGold.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
solc-0.8.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Variable ERC20._totalSupply (RORAGold.sol#261) is too similar to RORAGold.constructor(string,string,uint256,address,address,addre
ss,address,uint256).totalSupply_ (RORAGold.sol#443)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-are-too-similar
INFO:Detectors:
name() should be declared external:
        - ERC20.name() (RORAGold.sol#271-273)
symbol() should be declared external:
        - ERC20.symbol() (RORAGold.sol#275-277)
decimals() should be declared external:
        - ERC20.decimals() (RORAGold.sol#279-281)
        - RORAGold.decimals() (RORAGold.sol#471-473)
balanceOf(address) should be declared external:
        - ERC20.balanceOf(address) (RORAGold.sol#287-289)
approve(address,uint256) should be declared external:
        - ERC20.approve(address,uint256) (RORAGold.sol#301-305)
increaseAllowance(address,uint256) should be declared external:
        - ERC20.increaseAllowance(address,uint256) (RORAGold.sol#318-322)
decreaseAllowance(address,uint256) should be declared external:
        - ERC20.decreaseAllowance(address,uint256) (RORAGold.sol#324-333)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:RORAGold.sol analyzed (12 contracts with 75 detectors), 14 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Solidity Static Analysis

RORAGold.sol

Gas & Economy

Gas costs:

Gas requirement of function RORAGold.mint is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 1175:4:

Gas costs:

Gas requirement of function RORAGold.setTimeLock is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 1211:4:

Miscellaneous

Constant/View/Pure functions:

RORAGold.revokeRole(bytes32,address) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 1246:4:

Constant/View/Pure functions:

RORAGold.grantRole(bytes32,address) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 1251:4:

Similar variable names:

RORAGold.(string,string,uint256,address,address,address,address,uint256) : Variables have very similar names "_totalSupply" and "totalSupply_". Note: Modifiers are currently not considered by this static analysis.
Pos: 1160:31:

Similar variable names:

RORAGold.renounceRole(bytes32,address) : Variables have very similar names "_roles" and "role". Note: Modifiers are currently not considered by this static analysis.
Pos: 1241:16:

Similar variable names:

RORAGold.renounceRole(bytes32,address) : Variables have very similar names "_roles" and "role". Note: Modifiers are currently not considered by this static analysis.
Pos: 1242:24:

No return:

IERC20Metadata.decimals(): Defines a return type but never explicitly returns a value.
Pos: 713:4:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
more
Pos: 1252:8:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
more
Pos: 1257:8:

Solhint Linter

RORAGold.sol

```
RORAGold.sol:921:18: Error: Parse error: missing ';' at '{'
RORAGold.sol:954:18: Error: Parse error: missing ';' at '{'
RORAGold.sol:1003:18: Error: Parse error: missing ';' at '{'
RORAGold.sol:1054:22: Error: Parse error: missing ';' at '{'
```

Software analysis result:

These software reported many false positive results and some are informational issues. So, those issues can be safely ignored.

Email: audit@EtherAuthority.io