Arbitech Solutions

AUDIT REPORT

HURA TOKEN BEP20 SMART CONTRACT

Table of contents

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.

Introduction

Arbitech Solutions was performing the Security audit of the Pety 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 starting May 29th, 2023

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

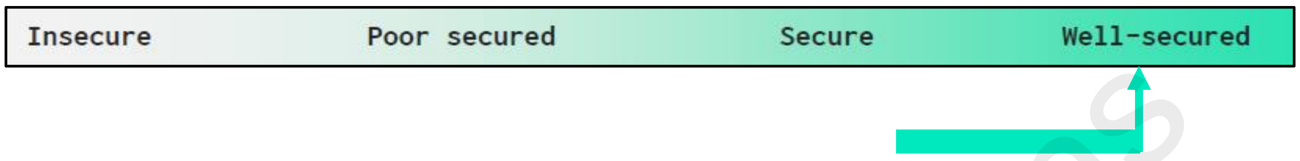
- The provided Solidity code appears to be a contract called "Pety." It is an ERC721 token contract with additional functionality for upgrading and minting pet creatures.
- The contract imports several other contracts from the OpenZeppelin library, including ERC721EnumerableUpgradeable and OwnableUpgradeable.

Audit scope

Name	**Code Review and Security Analysis Report for Pety Smart Contract**
Platform	**BSC / Solidity**
File	HuraToken.sol
Online code link	0xB6bE9EC2d52aa665488DC1E8aE6D306ec0475a1B
Audit Date	May 29th, 2023

Audit Summary

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



You are here

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 2 critical,

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	Moderate
	Features claimed	Passed
	Other programming issues	Passed
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Moderate
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 the Pety Smart Contract 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 Pety Smart Contract.

The Pety Smart Contract 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 well commented on in the smart contracts. Ethereum's NatSpec commenting style is used, which is a good thing.

Documentation

We were given a Pety smart contract code in the form of a Github web link. The hash of that code is mentioned above in the table.

As mentioned above, code parts are **not well** commented. But the logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

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 not used in external smart contract calls.

AS-IS overview

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	onlyModerator	modifier	Passed	No Issue
3	moderator	read	Passed	No Issue
4	mint	write	access only moderator	No Issue
5	burn	write	access only moderator	No Issue
6	changeModerator	write	access only moderator	No Issue
7	changeModeratorMaxSupply	write	access only moderator	No Issue
8	resetModeratorSupply	write	access only moderator	No Issue

Severity Definitions

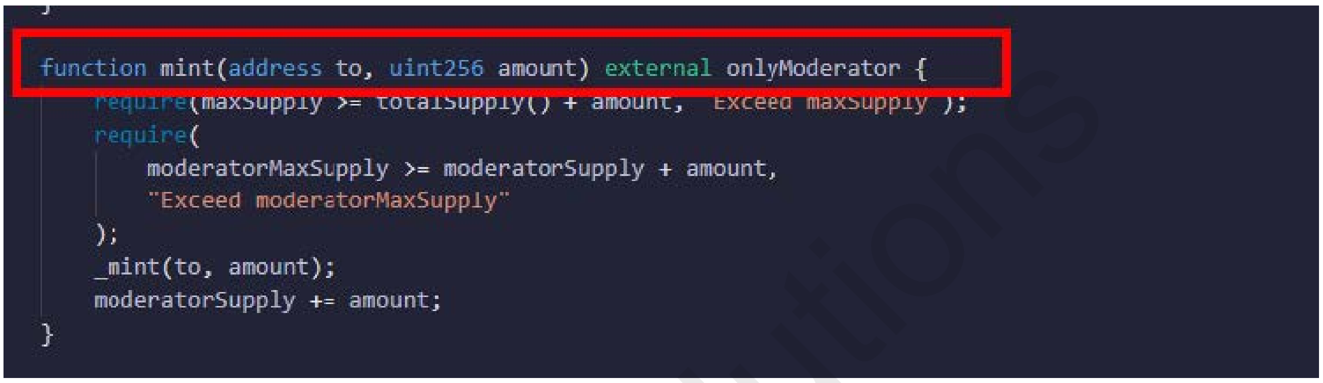
Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial
Medium	Medium-level 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.

Audit Findings

Certainly! Here are the detailed findings for the mentioned issues in the HuraToken contract:

Medium Severity

Function: mint(to, amount)



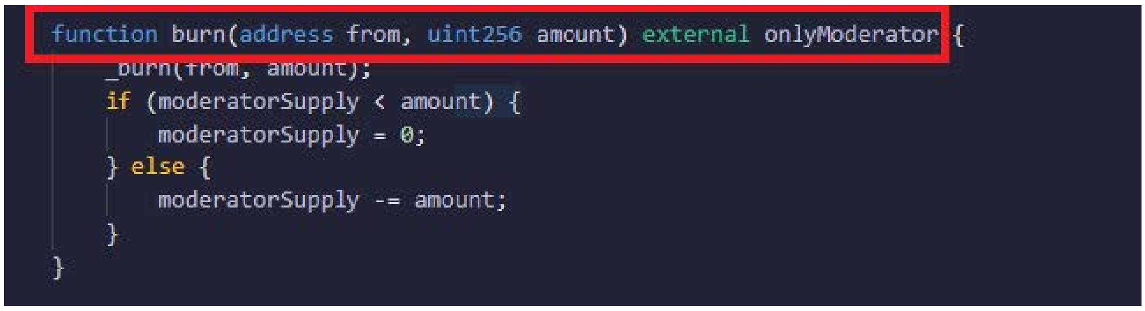
```
}

function mint(address to, uint256 amount) external onlyModerator {
    require(maxSupply >= totalSupply() + amount, "Exceed maxSupply");
    require(
        moderatorMaxSupply >= moderatorSupply + amount,
        "Exceed moderatorMaxSupply"
    );
    _mint(to, amount);
    moderatorSupply += amount;
}
```

The presence of a mint function controlled by the owner allows the owner to create new tokens at will.
This functionality raises concerns about potential inflationary practices or abuse of token supply.
Trust Percentage:
Dex tools or token explorers might display a lower trust percentage for tokens with a mint function controlled by the owner.

Resolution:
If the intention is to have a fixed supply token, remove or restrict the ability of the owner to mint new tokens.
Implement a minting mechanism during the token contract deployment phase, allowing the initial token supply to be created, but preventing any further minting.
Event issue:
Emit events within the mint function to provide transparency and allow external systems to track token minting activities.
Define an event, such as TokenMinted, and emit it with relevant information, such as the recipient's address and the amount of tokens minted.

Function : burn(from, amount)



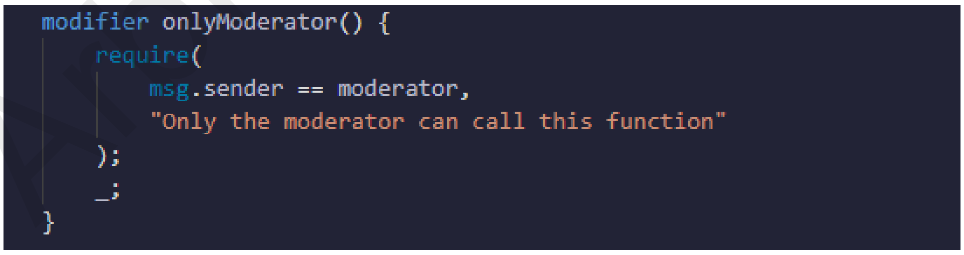
```
function burn(address from, uint256 amount) external onlyModerator {
    _burn(from, amount);
    if (moderatorSupply < amount) {
        moderatorSupply = 0;
    } else {
        moderatorSupply -= amount;
    }
}
```

The presence of a burn function controlled by the owner allows the owner to destroy or remove tokens from circulation.
This functionality raises concerns about potential deflationary practices or intentional token supply manipulation.
Dex tools or token explorers might display a lower trust percentage for tokens with a burn function controlled by the owner.

Resolution: Consider implementing additional checks or access controls to restrict the burning of tokens to specific addresses or roles.
Introduce role-based access control or multi-signature mechanisms to ensure that token burning actions are authorized and transparent.
Mint is not fixed supply:

Other Functionalities

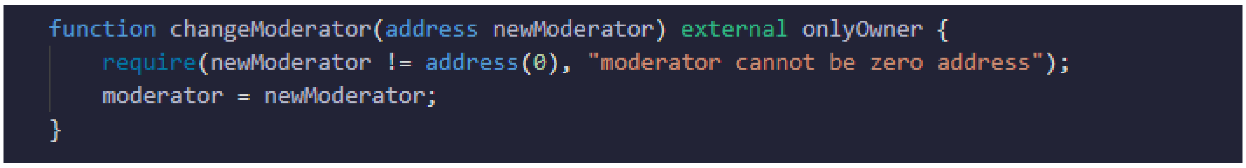
modifier onlyModerator()



```
modifier onlyModerator() {
    require(
        msg.sender == moderator,
        "Only the moderator can call this function"
    );
    _;
}
```

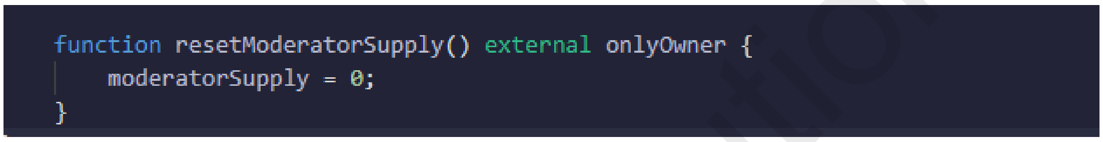
onlyModerator modifier adds a condition that only allows the function to be executed by the designated moderator

Function: changeModerator(address)


```
function changeModerator(address newModerator) external onlyOwner {
    require(newModerator != address(0), "moderator cannot be zero address");
    moderator = newModerator;
}
```

This function allows the contract owner to change the address of the moderator.

Function: changeModerator(address)


```
function resetModeratorSupply() external onlyOwner {
    moderatorSupply = 0;
}
```

This function allows the contract owner to change the maximum supply value for the moderator.

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:

HuraToken.sol

- mint()
- burn()
- changeModerator()
- changeModeratorMaxSupply()
- resetModeratorSupply()

Ownable.sol

- renounceOwnership: Deleting ownership will leave the contract without an owner, removing any owner-only functionality.
- transferOwnership: Current owner can transfer ownership of the contract to a new account.

Conclusion

We were given a contract code in the form of a GitHub and BSCScan link. we have used all possible tests based on given objects as files. We confirm that 2 high-severity issue**.**

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.

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

The security state of the reviewed smart contract, based on standard audit procedure scope, 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.

Disclaimers

Arbitech Solutions Disclaimer

Arbitech Solutions 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).

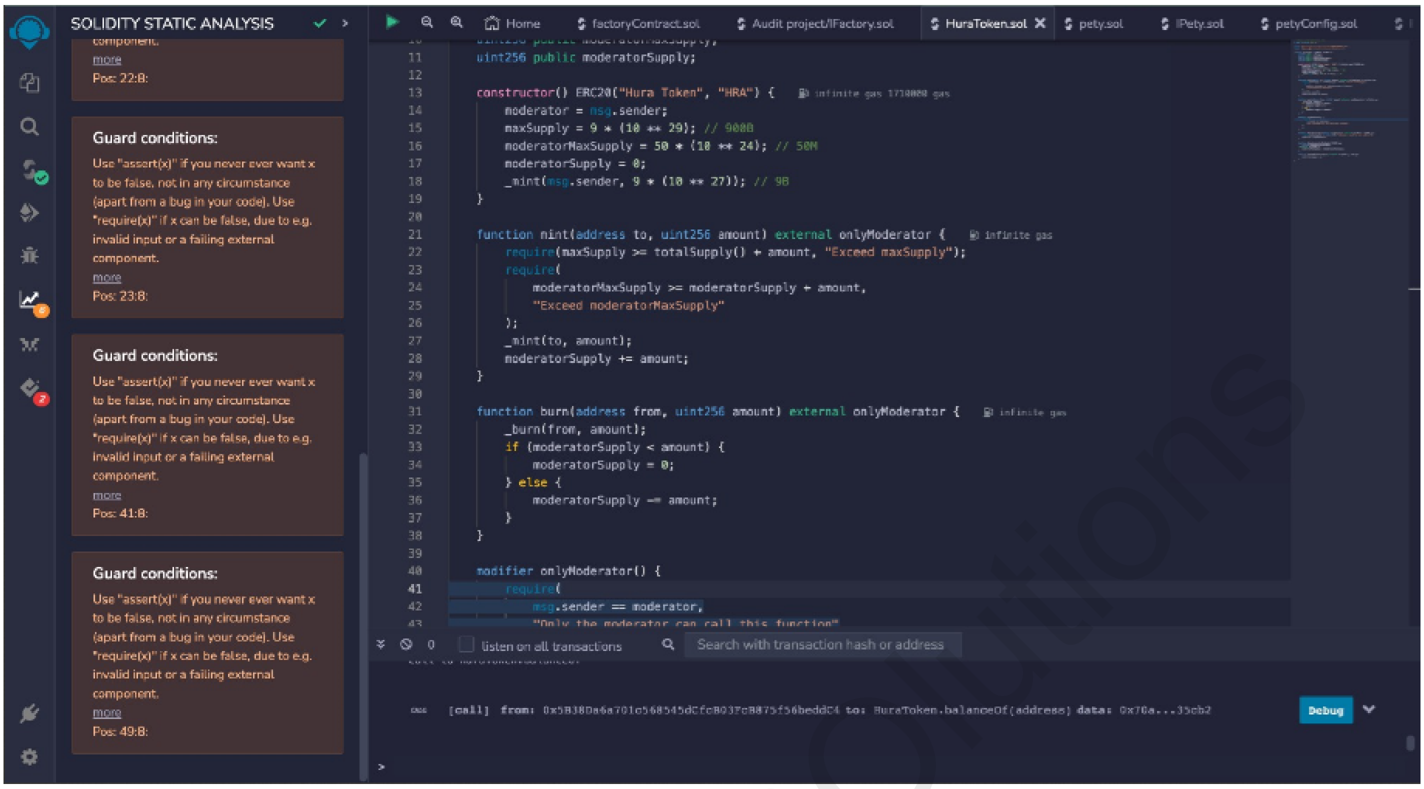
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While 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.

Solidity Static Analysis

HuraToken.sol



component.
more
Pos: 22:8:

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: 23:8:

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: 41:8:

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: 49:8:

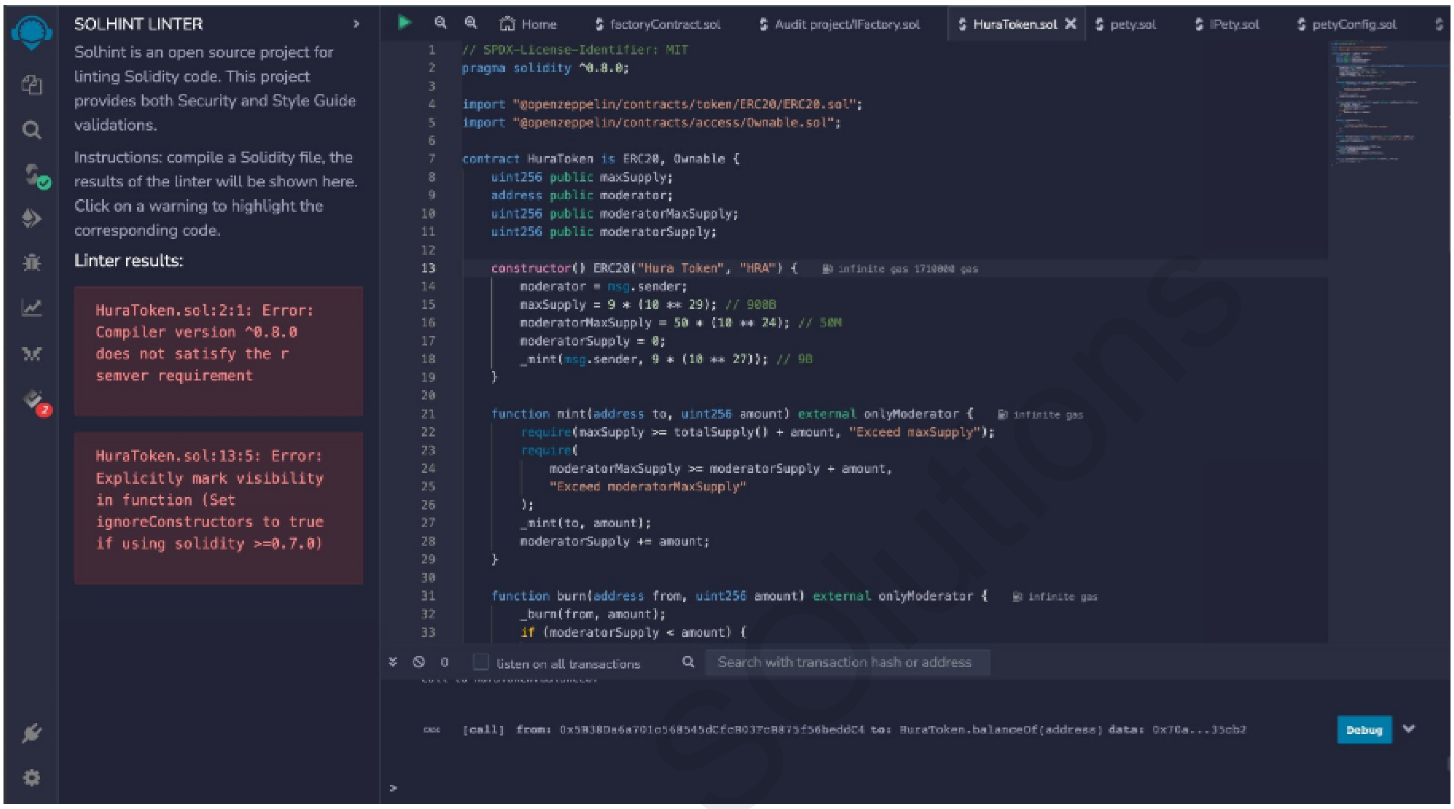
```
10      uint256 public moderatorMaxSupply;
11      uint256 public moderatorSupply;
12
13      constructor() ERC20("Hura Token", "HRA") {     infinite gas 1718000 gas
14          moderator = msg.sender;
15          maxSupply = 9 * (10 ** 29); // 900B
16          moderatorMaxSupply = 50 * (10 ** 24); // 50M
17          moderatorSupply = 0;
18          _mint(msg.sender, 9 * (10 ** 27)); // 9B
19      }
20
21      function mint(address to, uint256 amount) external onlyModerator {     infinite gas
22          require(maxSupply >= totalSupply() + amount, "Exceed maxSupply");
23          require(
24              moderatorMaxSupply >= moderatorSupply + amount,
25              "Exceed moderatorMaxSupply"
26          );
27          _mint(to, amount);
28          moderatorSupply += amount;
29      }
30
31      function burn(address from, uint256 amount) external onlyModerator {     infinite gas
32          _burn(from, amount);
33          if (moderatorSupply < amount) {
34              moderatorSupply = 0;
35          } else {
36              moderatorSupply -= amount;
37          }
38      }
39
40      modifier onlyModerator() {
41          require(
42              msg.sender == moderator,
43              "Only the moderator can call this function"
```

```
0    listen on all transactions    Search with transaction hash or address

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: HuraToken.balanceOf(address) data: 0x70a...35cb2    Debug
>
```

Solhint Linter

Hura.Token.sol



SOLHINT LINTER

Solhint is an open source project for linting Solidity code. This project provides both Security and Style Guide validations.

Instructions: compile a Solidity file, the results of the linter will be shown here. Click on a warning to highlight the corresponding code.

Linter results:

```
HuraToken.sol:2:1: Error:
Compiler version ^0.8.0
does not satisfy the r
semver requirement
```

```
HuraToken.sol:13:5: Error:
Explicitly mark visibility
in function (Set
ignoreConstructors to true
if using solidity >=0.7.0)
```

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract HuraToken is ERC20, Ownable {
    uint256 public maxSupply;
    address public moderator;
    uint256 public moderatorMaxSupply;
    uint256 public moderatorSupply;

    constructor() ERC20("Hura Token", "HRA") {    infinite gas 1710000 gas
        moderator = msg.sender;
        maxSupply = 9 * (10 ** 29); // 900B
        moderatorMaxSupply = 50 * (10 ** 24); // 50M
        moderatorSupply = 0;
        _mint(msg.sender, 9 * (10 ** 27)); // 9B
    }

    function mint(address to, uint256 amount) external onlyModerator {    infinite gas
        require(maxSupply >= totalSupply() + amount, "Exceed maxSupply");
        require(
            moderatorMaxSupply >= moderatorSupply + amount,
            "Exceed moderatorMaxSupply"
        );
        _mint(to, amount);
        moderatorSupply += amount;
    }

    function burn(address from, uint256 amount) external onlyModerator {    infinite gas
        _burn(from, amount);
        if (moderatorSupply < amount) {
```

```
[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: HuraToken.balanceOf(address) data: 0x70a...35cb2
```

Home
factoryContract.sol
Audit project/IFactory.sol
HuraToken.sol
pety.sol
IPety.sol
petyConfig.sol
listen on all transactions
Search with transaction hash or address
Debug

Software analysis result:

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