



Arbitech
Solutions

AUDIT REPORT

PETY SMART CONTRACT

Table of contents

Introduction	4
Project Background	4
Audit Scope	4
Audit Summary	5
Technical Quick Stats	6
Code Quality	7
Documentation	7
Use of Dependencies	7
AS-IS overview	8
Severity Definitions	9
Audit Findings	10
Conclusion	19
Our Methodology	21
Disclaimers	23
Appendix	
• Solidity static analysis	24
• Solhint Linter	25

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.

Arbitech Solutions

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	Pety.sol
Online code link	0x67fd91Fe3603Ebdc30f0B3DA3c79ad214F67b4bA
Audit Date	May 29th, 2023

Audit Summary

According to the standard audit assessment, Customer's solidity based smart contracts are **"Secured"**.



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.

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 3 smart contract. Smart contract contains Libraries, Smart contracts, inherits and Interfaces. These are compact and well written smart contracts.

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 and BSCScan web link.

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.

AS-IS overview

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	getToken()	read	Passed	No Issue
3	changeToken()	write	access only owner	No Issue
4	usdToNativeToken()	read	Passed	No Issue
5	usdToHuraToken()	read	Passed	No Issue
6	random()	read	Passed	No Issue
7	initialize()	write	Passed	No Issue
8	getPetyLevel()	read	Passed	No Issue
9	getTokenomic()	read	Passed	No Issue
9	getMaxLevel()	read	Passed	No Issue
10	changePrice()	modifier	access only Owner	No Issue
11	changeUpgradePrice()	modifier	access only owner	No Issue
12	changeUpgradeSuccessRate()	modifier	access only owner	No Issue
13	changeTokenomic()	modifier	access only owner	No Issue
14	mint()	modifier	access only owner	No Issue
15	upgrade()	write	Passed	No Issue
16	upgradeByHuraToken()	write	Passed	No Issue
17	buy()	write	Passed	No Issue
18	changeBaseURI()	modifier	access only owner	No Issue
19	changeTolerance()	modifier	access only owner	No Issue
20	comparePrice()	read	Passed	No Issue
21	contractURI()	read	Passed	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 Factory contract:

Low Severity

Function: random()

```
function random() external view returns (uint256) {  
    return  
        uint256(  
            keccak256(  
                abi.encodePacked(  
                    block.difficulty,  
                    block.timestamp,  
                    usdToNativeToken(1)  
                )  
            )  
        )  
};
```

FINDINGS IN FACTORY

Based on the provided code, it appears that the contract is using the keccak256 function to generate a random number. However, it is generally not recommended to rely on this method for generating random numbers, especially in scenarios where fairness and security are crucial. The keccak256 function is deterministic and can be manipulated by miners in certain situations.

SOLUTION:

For a more secure and tamper-resistant random number generation, it is advisable to use a verifiable random function (VRF). Chainlink VRF (Verifiable Random Function) is a widely recognized and trusted solution for generating random numbers on the blockchain. It utilizes external oracle services and cryptographic proofs to ensure the randomness and integrity of the generated numbers.

Reference: You can use this link to get verifiable random number.

<https://docs.chain.link/vrf/v2/introduction>

Other Functionalities in Factory Smart Contract

Function: getToken()

```
function getToken() public view returns (IERC20) {  
    return token;  
}
```

getToken() function provides a way for external callers to access the token contract instance stored in the token variable. By returning the token variable, external callers can interact with the ERC20 token contract and access its functions and properties.

Function: changeToken()

```
function changeToken(address _newToken) public onlyOwner {  
    token = IERC20(_newToken);  
}
```

Function: usdToNativeToken()

```
function usdToNativeToken(uint256 usd) public view returns (uint256) {  
    (  
        ,  
        /*uint80 roundID*/  
        int256 price /*uint startedAt*/ /*uint timeStamp*/ /*uint80 answeredInRound*/,  
        ,  
        ,  
    ) = priceFeed.latestRoundData();  
    uint256 result = (usd * (10 ** 26)) / uint256(price);  
    return result;  
}
```

The **usdToNativeToken()** function leverages the priceFeed from Chainlink to convert a USD value in wei to the equivalent value in native tokens. It uses the latest price obtained from the priceFeed and performs a simple calculation to determine the conversion.

Function: `usdToHuraToken()`

```
function usdToHuraToken(uint256 _price) public view returns (uint256) {
    address[] memory path = new address[](2);
    path[0] = usdAddress;
    path[1] = address(token);
    uint256[] memory amountOuts = router.getAmountsOut(_price * 1e18, path); // busd price = hura token amount
    return amountOuts[1];
}
```

The `usdToHuraToken()` function uses the Uniswap router (router) to determine the amount of Hura tokens that can be obtained by swapping a specified USD price. It constructs a token conversion path, performs the swap using the `getAmountsOut` function, and returns the resulting amount of Hura tokens.

Functionalities PetyConfig Smart Contracts

Function: `initialize()`

```
function initialize(
    uint32[] memory _prices,
    uint32[] memory _upgradePrices,
    uint8[11] memory _upgradeSuccessRates
) public initializer {
    for (uint8 i = 0; i <= 10; i += 1) {
        levels[i].price = _prices[i];
        levels[i].upgradePrice = _upgradePrices[i];
        levels[i].upgradeSuccessRate = _upgradeSuccessRates[i];
    }
    tokenomic = 80000; // 80%
    maxLevel = 10;
    __Ownable_init();
}
```

The `initialize()` function initializes the contract by setting the prices, upgrade prices, and success rates for each level. It also sets the tokenomic value, maximum level, and initializes the contract as an ownable contract. This function is typically called once during contract deployment to set the initial configuration of the contract.

Function: initialize()

```
function getPetyLevel(  
    uint8 _level  
) external view returns (Pety.Level memory) {  
    return levels[_level];  
}
```

The **getPetyLevel()** function provides a convenient way for external callers to retrieve the details of a specific Pety level by providing the level number as a parameter. By returning the corresponding Pety.Level struct from the levels array, external callers can access the properties of the level struct to obtain information about that particular Pety level.

Function: getMaxLevel()

```
function getMaxLevel() external view returns (uint32) {  
    return maxLevel;  
}
```

The **getMaxLevel()** function is an external view function that allows external callers to retrieve the maximum level of the contract.

Function: changePrice()

```
function changePrice(uint8 _level, uint32 newPrice) external onlyOwner {  
    require(_level <= 30, "level cannot greater than 30");  
    require(  
        _level <= 1 || levels[_level - 1].price != 0,  
        "level is invalid"  
    );  
    levels[_level].price = newPrice;  
    if (_level > maxLevel) {  
        maxLevel = _level;  
    }  
}
```

The **changePrice()** function allows the contract owner to change the price of a specific Pety level. It performs input validations, updates the price, and potentially updates the maxLevel if necessary. This function provides flexibility to adjust the prices of different levels of the Pety contract as desired by the contract owner.

Function: changeUpgradePrice()

```
function changeUpgradePrice(  
    uint8 _level,  
    uint32 _newUpgradePrice  
) external onlyOwner {  
    require(levels[_level].price != 0 || _level == 0, "level is invalid");  
    levels[_level].upgradePrice = _newUpgradePrice;  
}
```

The **changeUpgradePrice()** function allows the contract owner to change the upgrade price of a specific Pety level. It performs an input validation to ensure that the level is valid, and then updates the upgrade price of the level accordingly. This function provides flexibility to adjust the upgrade prices of different levels of the Pety contract as desired by the contract owner.

Function: changeUpgradeSuccessRate ()

```
function changeUpgradeSuccessRate(  
    uint8 _level,  
    uint8 newRate  
) external onlyOwner {  
    require(levels[_level].price != 0 || _level == 0, "level is invalid");  
    levels[_level].upgradeSuccessRate = newRate;  
}
```

The **changeUpgradeSuccessRate()** function allows the contract owner to change the upgrade success rate of a specific Pety level

Function: changeTokenomic ()

```
function changeTokenomic(uint32 newTokenomic) external onlyOwner {  
    tokenomic = newTokenomic;  
}
```

The **changeTokenomic()** function allows the contract owner to change the tokenomic value of the contract. It updates the tokenomic variable with the provided value, allowing the contract owner to adjust the tokenomic percentage as desired.

Functionalities PetyConfig Smart Contracts

Function: initialize()

```
function initialize(
    address factoryAddress,
    address petyConfigAddress
) public initializer {
    factory = IFactory(factoryAddress);
    petyConfig = IPetyConfig(petyConfigAddress);
    tolerance = 1000;
    baseURI = "https://petmapping.com/metadata/";
    __ERC721_init("Pety", "PETY");
    __Ownable_init();
}
```

The **initialize()** function is called during the contract deployment process to set up the initial state of the contract by providing the necessary contract addresses and initializing variables and contracts.

Function: mint()

```
function mint(uint8 _level, uint8 _numberOfPeties) public onlyOwner {
    uint256 _tokenId = totalSupply();
    for (uint8 i = 0; i < _numberOfPeties; i++) {
        level[_tokenId + i] = _level;
        _mint(msg.sender, _tokenId + i);
    }
}
```

The **mint()** function allows the owner of the contract to mint multiple Pety tokens. Each token is assigned a specified level and is minted with a unique token ID. This function provides a convenient way for the contract owner to create multiple Pety tokens at once.

Function: **upgradeByHuraToken()**

```
function upgradeByHuraToken(uint256 _tokenId) public {
    uint8 _level = level[_tokenId];
    Level memory petyLevel = petyConfig.getPetyLevel(_level);
    require(petyLevel.upgradePrice != 0, "Reached highest level");
    uint256 value = factory.usdToHuraToken(petyLevel.upgradePrice);
    value = (value * petyConfig.getTokenomic()) / 100000;
    require(
        factory.getToken().transferFrom(msg.sender, address(this), value),
        "Failed in transfer to contract"
    );
    require(
        factory.getToken().transfer(owner(), value),
        "Failed in transfer to Pet Mapping"
    );
    _upgrade(_tokenId, _level);
}
```

The **upgradeByHuraToken()** function allows the token holder to upgrade their Pety token using Hura tokens. It retrieves the current level of the token and obtains the upgrade price for the next level from the Pety configuration. The upgrade price is converted from USD to Hura tokens using the factory's conversion function. User will get 20% discount if he will upgrade level by using HuraToken.

Then, the function transfers the required amount of Hura tokens from the caller to the contract. The transferred tokens are then immediately transferred from the contract to the owner of the Pety contract. Finally, the `_upgrade` function is called to perform the upgrade process, updating the level of the token if the upgrade is successful.

Function: **changeTolerance()**

```
function changeTolerance(uint32 _newValue) public onlyOwner {
    tolerance = _newValue;
}
```

The **changeTolerance()** function provides flexibility for the contract owner to adjust the tolerance level used in price comparisons based on their specific requirements or market conditions.

Function: **buy ()**

```
function buy(uint256 _tokenId) public payable mutex(_tokenId) {
    require(ownerOf(_tokenId) == owner(), "This NFT is unavailable");
    uint8 _level = level[_tokenId];
    Level memory petyLevel = petyConfig.getPetyLevel(_level);
    require(
        comparePrice(msg.value, factory.usdToNativeToken(petyLevel.price)),
        "You need to pay ether"
    );
    (bool transferred, ) = owner().call{value: msg.value}("");
    require(transferred, "Transfer failed");
    _transfer(owner(), msg.sender, _tokenId);
}
```

The `buy` function allows a user to purchase a Pety token by sending Ether (ETH) to the contract. It takes the token ID as a parameter.

First, the function checks if the owner of the token is the contract owner. This ensures that the token is available for purchase.

Next, it retrieves the current level of the token from the `level` mapping.

Then, it fetches the pricing information for the token's level from the Pety configuration.

The function compares the value sent in the transaction (`msg.value`) with the price of the token in native currency (USD) using the `comparePrice` function. If the sent value is not equal to or within the acceptable tolerance of the token price, the transaction reverts with an error message.

If the sent value is correct, the function transfers the received Ether to the contract owner using the `owner().call{value: msg.value}("")` expression. This transfer is executed using a low-level `call` to prevent reentrant attacks.

Finally, if the transfer is successful, the function transfers the ownership of the token from the contract owner to the buyer using the `_transfer` function inherited from ERC721.

Function: **changeBaseURI ()**

```
function changeBaseURI(string memory newBaseURI) external onlyOwner {
    baseURI = newBaseURI;
}
```

The changeBaseURI function allows the contract owner to change the base URI for the token metadata. It takes a new base URI as a parameter and updates the baseURI variable with the new value.

Function: **tokenURI ()**

```
function tokenURI(  
    uint256 tokenId  
) public view virtual override returns (string memory) {  
    string memory _level = Strings.toString(level[tokenId]);  
    return string(abi.encodePacked(baseURI, _level, ".json"));  
}
```

The **tokenURI()** function is an overridden function from the ERC721 standard. It is used to generate the URI for a specific token's metadata.

Function: **contractURI ()**

```
function contractURI() public view returns (string memory) {  
    return string(abi.encodePacked(baseURI, "pety.json"));  
}
```

The **contractURI()** function returns the URI for the contract's metadata. It concatenates the baseURI with the string "pety.json" to form the complete URI.

Function: **comparePrice ()**

```
function comparePrice(uint256 a, uint256 b) public view returns (bool) {  
    if (a == b) {  
        return true;  
    }  
    uint256 subtract;  
    uint256 max;  
    if (a < b) {  
        subtract = b - a;  
        max = b;  
    } else {  
        subtract = a - b;  
        max = a;  
    }  
    return subtract <= ((max * tolerance) / 100000);  
}
```

This function can be used to compare prices and check if they are approximately equal within a certain tolerance level, which can be useful in scenarios where precise equality is not required but a close approximation is sufficient.

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

**SOLIDITY STATIC ANALYSIS** ✓ >

Last results for:
Pety.sol


☐ Show warnings for external libraries


Security

Check-effects-interaction:
Potential violation of Checks-Effects-Interaction pattern in
Pety.upgrade(uint256): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.
[more](#)
Pos: 45:4:


Check-effects-interaction:
Potential violation of Checks-Effects-Interaction pattern in
Pety.upgradeByHuraToken(uint256): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.
[more](#)
Pos: 61:4:

Check-effects-interaction:
Potential violation of Checks-Effects-


petyConfig.sol

Pety.sol X

```
4 import "@openzeppelin/contracts-upgradeable/token/ERC721/extensions/ERC721EnumerableUpgradeable.sol";
5 import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
6 import "../interfaces/IFactory.sol";
7 import "@openzeppelin/contracts/utils/Strings.sol";
8 import "../interfaces/IPetyConfig.sol";
9
10 contract Pety is ERC721EnumerableUpgradeable, OwnableUpgradeable {
11     struct Level {
12         uint32 price; // price for each level by USD
13         uint32 upgradePrice;
14         uint8 upgradeSuccess
15     }
16     IFactory public factory;
17     IPetyConfig public petyConfig;
18     mapping(uint256 => uint8) public level;
19     event Upgrade(uint256 tokenId, bool success);
20     mapping(uint256 => bool) public locked;
21     uint32 public tolerance;
22     string public baseURI;
23
24     function initialize(
25         address factoryAddress,
26         address petyConfigAddress
27     ) public initializer {
28         factory = IFactory(factoryAddress);
29         petyConfig = IPetyConfig(petyConfigAddress);
30         tolerance = 1000;
31         baseURI = "https://petmapping.com/metadata/";
32         __ERC721_init("Pety", "PETV");
33         __Ownable_init();
34     }
```

0

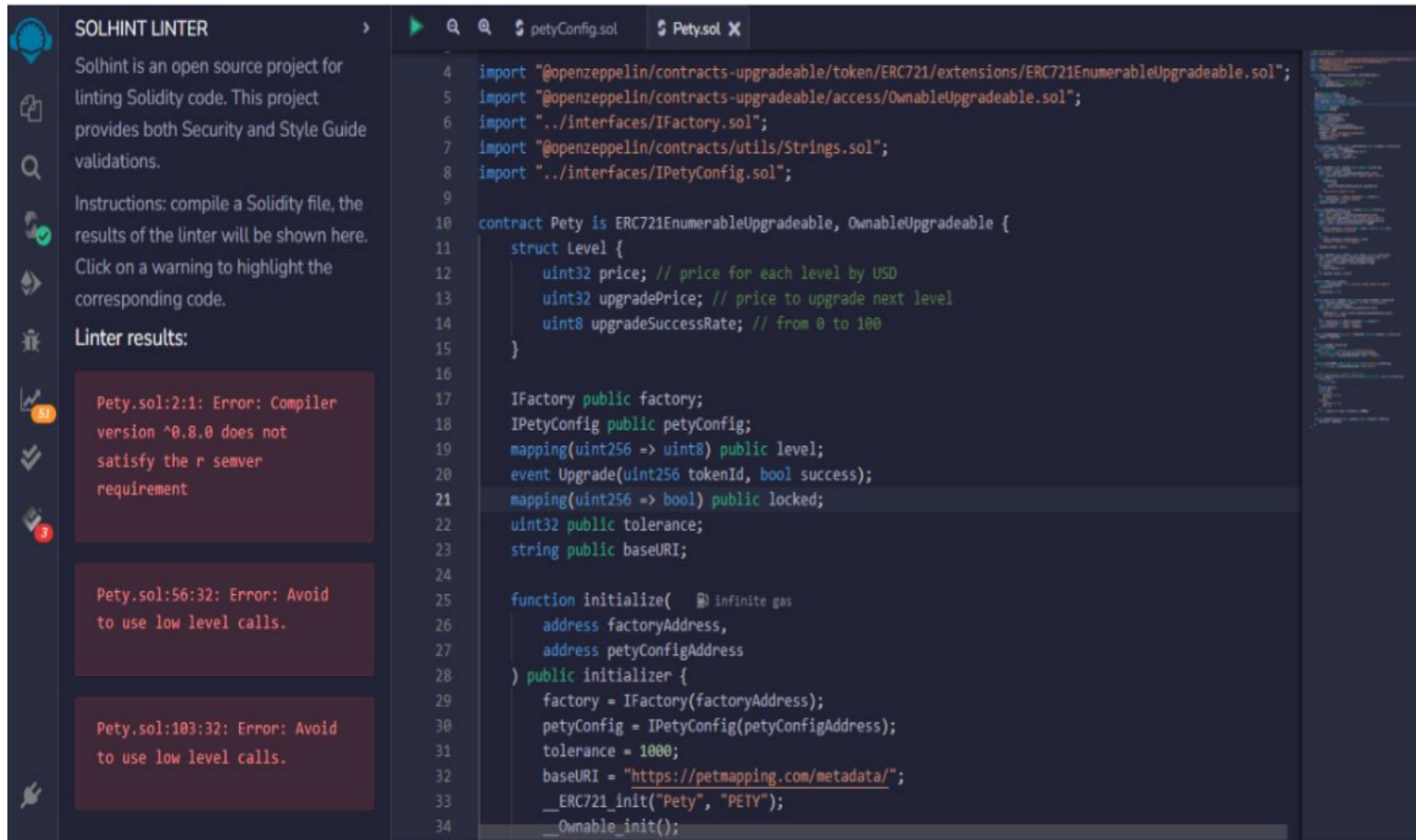
☐ listen on all transactions

 Search with transaction hash or address

Arbitech Solutions

Solhint Linter

Hura.Token.sol



Software analysis result:

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