Lesbian Inu Token

Security Audit

May 26, 2023,

V 1.0.0

Introduction	3
Additional Info about Audited Project	4
Source Code	5
Methodology	6
Issues Descriptions and Recommendations	7
Report	8
Functions (Read & Write)	12
Disclaimer	13

This document includes the results of the security audit for smart contract (0x15a133bA390FfD210C13a03950F0D2dFe6E14B84 on BSC chain,
code as found in the section titled 'Source Code'. our security team performed the security audit from May 25, 2023, to May 26, 2023. The purpose of this audit is to review the source code of specific Solidity contracts and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer:

While our team review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

We identified some low to high-severity issues.

Specification

Our understanding of the specification was based on the following sources:

- The Source code found on the block-chain explorer.

Additional Info about Audited Project

Contract Name - Lesbian Inu
Contract Type - ERC20
Contract Add. - 0x15a133bA390FfD210C13a03950F0D2dFe6E14B84
Block-chain explorer link - link
Compiler Version - v0.8.2
License - not available
Decimal - 18
Total Supply - 1,000,000,000,000,000

Contract created by - 0xe50Fd6c76D0344eD76AeC59c98824d34cCDBf541

Source Code

the following source code was reviewed during the audit:

Link:- Source Code

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves and does not pertain to any other programs or scripts, including deployment scripts and libraries.

Libraries used in Contract

None

Methodology

The audit was conducted in several steps.

First, we reviewed in detail all available documentation and specifications for the project, as described in the ‘Specification’ section above.

Second, we performed a thorough manual review of the code, checking that the code matched the specification and the spirit of the contract (i.e. the intended behavior). During this manual review portion of the audit, we primarily searched for security vulnerabilities, unwanted behavior vulnerabilities, and problems with systems of incentives.

Third, we performed the automated portion of the review consisting of measuring test coverage (while also assessing the quality of the test suite) and evaluating the results of various symbolic execution tools against the code.

Lastly, we performed a final line-by-line inspection of the code – including comments –in an effort to find any minor issues with code quality, documentation, or best practices.

Issues Descriptions and Recommendations

Severity Level Reference:

Level	Description
High	The issue poses an existential risk to the project, and the issue identified could lead to massive financial or reputational repercussions. We highly recommend fixing the reported issue. If you have already deployed, you should upgrade or redeploy your contracts.
Medium	The potential risk is large, but there is some ambiguity surrounding whether or not the issue would practically manifest. We recommend considering a fix for the reported issue.
Low	The risk is small, unlikely, or not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
Code Quality	The issue identified does not pose any obvious risk, but fixing it would improve overall code quality, conform to Possible best practices, and perhaps lead to fewer development issues in the future.
Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Based on these Levels we have found some particular issues in the smart contract after we have found after checking all parameters and given information.

Upon reviewing the provided Solidity code, I have identified three potential vulnerabilities related to the approve() function in the Token contract.

Here's issue 1:

In the approve() function, the allowance mapping is directly updated without considering the existing allowance value.

```
allowance[msg.sender][spender] = value;
```

This means that each time the approve() function is called, it sets the allowance value to the new value specified. However, it does not consider or update the previous allowance value, which can lead to unexpected behavior.

This vulnerability can be exploited in the following way:

1. Assume there are two parties, A and B.
2. Party A has tokens and approves Party B to spend a certain amount of tokens.
3. Before Party B utilizes the approved allowance, Party A decides to call the approve() function again with a different value, possibly reducing the allowance.
4. As a result, Party B still has the previously approved allowance but can now spend more tokens than intended by Party A.

To mitigate this vulnerability, you can modify the approve() function to handle the allowance updates properly, taking into account the existing allowance value. One common approach is to use the SafeMath library to prevent overflows/underflows when updating the allowance.

Possible Solution:

Here's an example of how you can modify the approve() function to address the vulnerability using SafeMath:

```
import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract Token {
    using SafeMath for uint256;

    // ... Contract code ...

    function approve(address spender, uint256 value) public returns (bool) {
        require(spender != address(0));

        allowance[msg.sender][spender] = allowance[msg.sender][spender].add(value);
        emit Approval(msg.sender, spender, allowance[msg.sender][spender]);

        return true;
    }
```

By utilizing SafeMath's add() function, the updated allowance value is correctly calculated, considering the existing allowance.

Here's issue 2:

Upon further review, I have identified another potential vulnerability in the provided Solidity code:

In the transferFrom() function, there is no check to ensure that the spender has sufficient allowance to transfer the specified value from the "from" address.

```
allowance[from][msg.sender] -= value;
balances[to] += value;
balances[from] -= value;
emit Transfer(from, to, value);
```

This means that anyone can call the transferFrom() function and transfer tokens from one address to another, even if they do not have the necessary allowance. This can lead to unauthorized token transfers.

To address this vulnerability, you should add a check to verify that the spender has sufficient allowance before executing the transfer. You can do this by comparing the allowance with the value being transferred.

Here's an example of how you can modify the transferFrom() function to include the allowance check:

Possible Solution:

```
function transferFrom(address from, address to, uint value) public returns(bool) {
    require(to != address(0));
    require(allowance[from][msg.sender] >= value); // Check if spender has sufficien

    allowance[from][msg.sender] -= value;
    balances[to] += value;
    balances[from] -= value;
    emit Transfer(from, to, value);
    return true;
}
```

By adding the `require(allowance[from][msg.sender] >= value)` statement, the function ensures that the spender has enough allowance to transfer the specified value from the "from" address. This helps prevent unauthorized transfers.

Here's issue 3:

1. The balanceOf() function does not specify the view modifier.

```
function balanceOf(address owner) public returns (uint) {
    return balances[owner];
}
```

Since the function only retrieves the balance of an address without modifying any state variables, it is best practice to add the view modifier to explicitly indicate that the function does not modify the contract's state.

Possible Solution:

To address this, you can modify the balanceOf() function as follows:

```
function balanceOf(address owner) public view returns (uint) {
    return balances[owner];
}
```

By adding the view modifier, the function explicitly states that it only reads from the contract's state and does not modify it. This helps improve code clarity and prevents any accidental modifications to the state when interacting with the function.

It is important to note that this vulnerability does not pose a security risk but is more of a best practice for code readability and maintenance.

Note - We have only checked and audited the contract (0x15a133bA390FfD210C13a03950F0D2dFe6E14B84).

All Functions of Smart Contract

READ Functions

1. Balance Of - shows the token balance of the wallet
2. Allowance - check allowance of a particular wallet
3. Decimals - shows the decimal quantity of token
4. Name - Name of contract
5. Symbol - Symbol of contract
6. Total supply - shows the total quantity of tokens

Write Functions

1. Transfer / Transfer From - for transferring token
2. Approve - for approving token for transfer/ trading.
3. Balance of - check the balance of wallet address

Summary

Smart Contract do not contain any severity issues.

Note
Please check the disclaimer above and note, the audit makes no statement or warranties models, investment attractiveness or code sustainability. The reports provide for only contract mention in the report