Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	Loveswap Token
Website:	Loveswap.com
Platform:	Binance Smart Chain
Language:	Solidity
Date:	October 29th, 2021

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 Loveswap team to perform the Security audit of the Loveswap Token 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 October 29th, 2021.

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

LoveSwap token is a BEP20 standard token smart contract running on Binance Smart Chain. This token serves as a backbone of the Loveswap Dex ecosystem. This technical audit scope only covers Loverswap Token smart contract only, and does not cover any other smart contracts in the Loveswap Protocol.

Audit scope

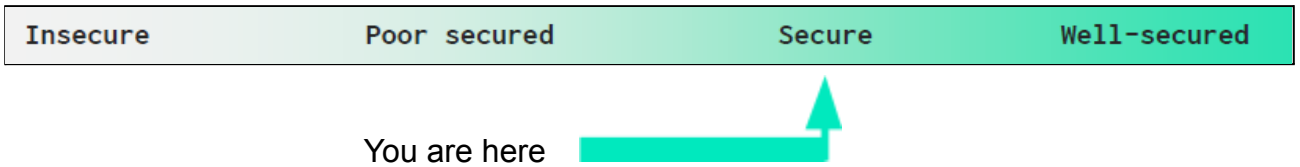
Name	**Code Review and Security Analysis Report for Loveswap Token Smart Contract**
Platform	**BSC / Solidity**
File	Loveswap.sol
File MD5 Hash	15A2A1A719DD790564E66127B434C183
Contract Address	0x8baf35803b452836a05A3e01ac36F3DDbF98bbE8
Audit Date	October 29th, 2021

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: ● Name: Loveswap DEX ● Symbol: Loveswap ● Decimals: 8 ● Total Supply: 1 trillion ● Token Minting: No more token minting ● Maximum Airdrop: 10,000,000	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, Customer`s solidity smart contracts are **“Secured”**. This token contract does not contain owner control, which does 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, 0 high, 0 medium and 2 low and some very low level issues. These issues are not critical ones, so it's good to go for the production.

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	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Moderated
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Moderated
	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	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	Passed
Gas Optimization	“Out of Gas” Issue	Passed
	High consumption ‘for/while’ loop	Moderated
	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**

Email: audit@EtherAuthority.io

Code Quality

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

The libraries in Loveswap 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 Loveswap Token.

The Loveswap 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.

Some code parts are **not** well commented on smart contracts.

Documentation

We were given a Loveswap Token smart contracts code in the form of the files. The hash of that code is mentioned above in the table.

As mentioned above, some code parts are not well commented, but most parts are commented. 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.

Another source of information was its official website https://www.loveswap.com which provided rich information about the project architecture and tokenomics.

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	totalSupply	read	Passed	No Issue
3	decimals	read	Passed	No Issue
4	editMaxAirdrop	write	Passed	No Issue
5	editAdmin	write	Passed	No Issue
6	claimAdmin	write	Passed	No Issue
7	airdrop	write	Possibility of heavy gas cost due to infinite loop	Owner must input limited wallets in airdrop
8	_airdrop	internal	Passed	No Issue
9	name	read	Passed	No Issue
10	symbol	read	Passed	No Issue
11	decimals	read	Passed	No Issue
12	totalSupply	read	Passed	No Issue
13	balanceOf	read	Passed	No Issue
14	transfer	write	Passed	No Issue
15	allowance	read	Passed	No Issue
16	approve	write	Passed	No Issue
17	transferFrom	write	Passed	No Issue
18	increaseAllowance	write	Passed	No Issue
19	decreaseAllowance	write	Passed	No Issue
20	_transfer	internal	Passed	No Issue
21	_approve	internal	Passed	No Issue
22	burn	write	Passed	No Issue
23	_mint	internal	Passed	No Issue
24	_beforeTokenTransfer	internal	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

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

(1) High gas consuming loop in airdrop function

```
function airdrop(address[] memory addresses, uint[] memory amounts) public {
    require(msg.sender == _admin, "Admin address required.");
    require(
        addresses.length == amounts.length,
        "Addresses and amounts arrays do not match in length."
    );
    for (uint i = 0; i < addresses.length; i++) {
        _airdrop(addresses[i], amounts[i] * 10**DECIMALS);
    }
```

The airdrop function allows the owner to input unlimited wallets. So, the owner must input limited wallets, as inputting excessive wallets might hit the block's gas limit. The owner can accept this risk and can execute this function using limited wallets only.

Resolution: We suggest specifying some limit on the number of wallets can be used. This will prevent any potential human error.

(2) No fractional value possible

```
function airdrop(address[] memory addresses, uint[] memory amounts) public {
    require(msg.sender == _admin, "Admin address required.");
    require(
        addresses.length == amounts.length,
        "Addresses and amounts arrays do not match in length."
    );
    for (uint i = 0; i < addr
        _airdrop(addresses[i], amounts[i] * 10**DECIMALS);
    }
```

In an airdrop function, the owner can not specify any fractional value like 0.05. The owner only has to specify only the whole numbers such as 100. This is because the owner specified amount is again multiplied with decimals.

Resolution: Ideally, the amount must be provided by the owner in the full decimal form. And that can be passed directly in the _airdrop function. This way the owner also can be able to specify any fractional value in airdrop.

Very Low / Informational / Best practices:

(1) Please make variables constant

```
string private _name;
string private _symbol;
uint private _totalSupply;
```

These variables' values will be unchanged. So, please make it constant. It will save some gas. Just put a constant keyword.

(2) Please use the latest compiler version when deploying contract

```
v0.8.0+commit.c7dfd78e
```

This is not a severe issue, but we suggest using the latest compiler version at the time of contract deployment, which is 0.8.9 at the time of this audit. Using the latest compiler version is always recommended which prevents any compiler level issues.

(3) All functions which are not called internally, must be declared as external. It is more efficient as sometimes it saves some gas.
https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices

(4) Approve of ERC20 / BEP20 standard:
To prevent attack vectors regarding approve() like the one described here:
https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. THOUGH the contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before.

Conclusion

We were given a contract code. And we have used all possible tests based on given objects as files. We observed some issues in the smart contracts, but they are not critical ones. So, **it's good to go to 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 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).

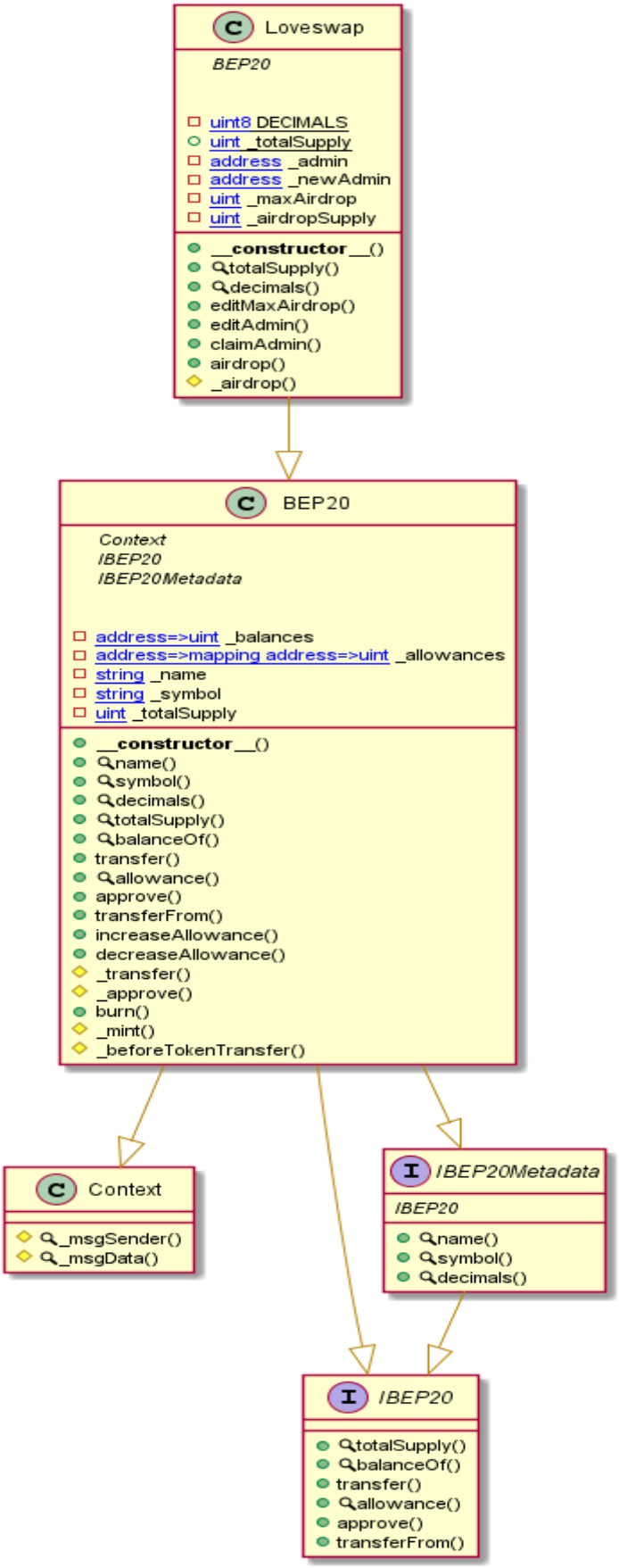
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we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



Email: audit@EtherAuthority.io

Slither Results Log

Slither log >> Loveswap.sol

```
INFO:Detectors:
Loveswap._totalSupply (Loveswap.sol#185) shadows:
        - BEP20._totalSupply (Loveswap.sol#50)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variable-shadowing
INFO:Detectors:
Loveswap.editMaxAirdrop(uint256) (Loveswap.sol#206-209) should emit an event for:
        - _maxAirdrop = newMax * 10 ** DECIMALS (Loveswap.sol#208)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic
INFO:Detectors:
Loveswap.editAdmin(address).newAdmin (Loveswap.sol#212) lacks a zero-check on :
                - _newAdmin = newAdmin (Loveswap.sol#214)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Context._msgData() (Loveswap.sol#9-12) 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 (Loveswap.sol#2) 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:
Constant Loveswap._totalSupply (Loveswap.sol#185) is not in UPPER_CASE_WITH_UNDERSCORES
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Redundant expression "this (Loveswap.sol#10)" inContext (Loveswap.sol#4-13)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
name() should be declared external:
        - BEP20.name() (Loveswap.sol#57-59)
symbol() should be declared external:
        - BEP20.symbol() (Loveswap.sol#61-63)
decimals() should be declared external:
        - BEP20.decimals() (Loveswap.sol#65-67)
        - Loveswap.decimals() (Loveswap.sol#202-204)
totalSupply() should be declared external:
        - BEP20.totalSupply() (Loveswap.sol#69-71)
        - Loveswap.totalSupply() (Loveswap.sol#198-200)
balanceOf(address) should be declared external:
        - BEP20.balanceOf(address) (Loveswap.sol#73-75)
```

```
transfer(address,uint256) should be declared external:
        - BEP20.transfer(address,uint256) (Loveswap.sol#77-80)
allowance(address,address) should be declared external:
        - BEP20.allowance(address,address) (Loveswap.sol#82-84)
approve(address,uint256) should be declared external:
        - BEP20.approve(address,uint256) (Loveswap.sol#86-89)
transferFrom(address,address,uint256) should be declared external:
        - BEP20.transferFrom(address,address,uint256) (Loveswap.sol#91-103)
increaseAllowance(address,uint256) should be declared external:
        - BEP20.increaseAllowance(address,uint256) (Loveswap.sol#105-108)
decreaseAllowance(address,uint256) should be declared external:
        - BEP20.decreaseAllowance(address,uint256) (Loveswap.sol#110-120)
burn(uint256) should be declared external:
        - BEP20.burn(uint256) (Loveswap.sol#152-163)
editMaxAirdrop(uint256) should be declared external:
        - Loveswap.editMaxAirdrop(uint256) (Loveswap.sol#206-209)
editAdmin(address) should be declared external:
        - Loveswap.editAdmin(address) (Loveswap.sol#212-215)
claimAdmin() should be declared external:
        - Loveswap.claimAdmin() (Loveswap.sol#219-222)
airdrop(address[],uint256[]) should be declared external:
        - Loveswap.airdrop(address[],uint256[]) (Loveswap.sol#226-235)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:Loveswap.sol analyzed (5 contracts with 75 detectors), 24 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
root@server:/chetan/gaza/mycontracts#
```

Solidity Static Analysis

Loveswap.sol

Gas & Economy

Gas costs:

Gas requirement of function BEP20.name 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: 57:4:

Gas costs:

Gas requirement of function Loveswap.name 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: 57:4:

Gas costs:

Gas requirement of function BEP20.symbol 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: 61:4:

Gas costs:

Gas requirement of function BEP20.burn 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: 152:4:

Gas costs:

Gas requirement of function Loveswap.burn 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: 152:4:

Gas costs:

Gas requirement of function Loveswap._totalSupply 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: 185:4:

Gas costs:

Gas requirement of function Loveswap.totalSupply 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: 198:4:

Gas costs:

Gas requirement of function Loveswap.editMaxAirdrop 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: 206:4:

Gas costs:

Gas requirement of function Loveswap.airdrop 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: 226:4:

For loop over dynamic array:

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point.
Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.
more
Pos: 232:8:

Miscellaneous

Constant/View/Pure functions:

IBEP20.transfer(address,uint256) : Potentially should be constant/view/pure but is not.
more
Pos: 20:4:

Constant/View/Pure functions:

IBEP20.approve(address,uint256) : Potentially should be constant/view/pure but is not.
more
Pos: 24:4:

Constant/View/Pure functions:

IBEP20.transferFrom(address,address,uint256) : Potentially should be constant/view/pure but is not.
more
Pos: 26:4:

Constant/View/Pure functions:

BEP20._beforeTokenTransfer(address,address,uint256) : Potentially should be constant/view/pure but is not.
more
Pos: 175:4:

Similar variable names:

BEP20.burn(uint256) : Variables have very similar names "account" and "amount".
Pos: 153:8:

Similar variable names:

BEP20.burn(uint256) : Variables have very similar names "account" and "amount".
Pos: 155:29:

Similar variable names:

BEP20.burn(uint256) : Variables have very similar names "account" and "amount".
Pos: 155:50:

Similar variable names:

BEP20.burn(uint256) : Variables have very similar names "account" and "amount".
Pos: 157:40:

Similar variable names:

BEP20.burn(uint256) : Variables have very similar names "account" and "amount".
Pos: 162:22:

Similar variable names:

BEP20.burn(uint256) : Variables have very similar names "account" and "amount".
Pos: 162:43:

Similar variable names:

BEP20._mint(address,uint256) : Variables have very similar names "account" and "amount".
Pos: 166:16:

Similar variable names:

BEP20._mint(address,uint256) : Variables have very similar names "account" and "amount".
Pos: 168:41:

Similar variable names:

BEP20._mint(address,uint256) : Variables have very similar names "account" and "amount".
Pos: 168:50:

Similar variable names:

BEP20._mint(address,uint256) : Variables have very similar names "account" and "amount".
Pos: 170:24:

No return:

IBEP20.approve(address,uint256): Defines a return type but never explicitly returns a value.
Pos: 24:4:

No return:

IBEP20.transferFrom(address,address,uint256): Defines a return type but never explicitly returns a value.
Pos: 26:4:

No return:

IBEP20Metadata.name(): Defines a return type but never explicitly returns a value.
Pos: 36:4:

No return:

IBEP20Metadata.symbol(): Defines a return type but never explicitly returns a value.
Pos: 38:4:

No return:

IBEP20Metadata.decimals(): Defines a return type but never explicitly returns a value.
Pos: 40:4:

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: 99:8:

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: 116:8:

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: 127:8:

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: 128:8:

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: 207:8:

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: 213:8:

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: 220:8:

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: 227:8:

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: 228:8:

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: 238:8:

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: 239:8:

Solhint Linter

Loveswap.sol

```
Loveswap.sol:2:1: Error: Compiler version ^0.8.0 does not satisfy the
r semver requirement
Loveswap.sol:52:5: Error: Explicitly mark visibility in function (Set
ignoreConstructors to true if using solidity >=0.7.0)
Loveswap.sol:179:24: Error: Code contains empty blocks
Loveswap.sol:185:26: Error: Constant name must be in capitalized
SNAKE_CASE
Loveswap.sol:192:5: Error: Explicitly mark visibility in function
(Set ignoreConstructors to true if using solidity >=0.7.0)
```

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