EtherAuthority

etherauthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	LSD
Website:	l7.finance
Platform:	Base Chain Network
Language:	Solidity
Date:	June 17th, 2024

Table of contents

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the LSD smart contract from l7.finance was audited extensively. 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 17th, 2024.

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

Website Details

- L7 Finance is a Web3 digital asset management and traffic aggregation platform. It offers a range of services including Centralized Exchanges (CEX), Decentralized Exchanges (DEX), crypto cards, and financial management solutions. The platform focuses on providing comprehensive digital asset investment and allocation services globally, engaging in venture capital, project incubation, and more.
- The native token of L7 Finance, LSD, is a BEP-20 token on the BNB blockchain. It provides various benefits such as discounted transaction fees on its trading platform, participation in liquidity pool mining, airdrop rights, and significant governance roles within the ecosystem.

Code Details

- The provided code is a Solidity smart contract implementing an ERC20 token named LSD. Here's a summary of the key components and functionalities:
- Key Components:
    - Interfaces and Inheritance:
        - IERC20: Interface defining the standard ERC20 functions.
        - Context: Provides information about the current execution context, such as the sender of the transaction and its data.
        - ERC20: Implements the IERC20 interface and provides the core functionality for ERC20 tokens.
        - ERC20Burnable: Extends ERC20 to add functions that allow token holders to destroy their own tokens or those that they have an allowance for.
    - Token Details:
        - name and symbol: Set during contract deployment and are immutable.
        - decimals: Defaults to 18, typical for most tokens, and can be overridden.
    - Core ERC20 Functions:
        - totalSupply(): Returns the total supply of the token.
        - balanceOf(address account): Returns the token balance of the specified address.
        - transfer(address recipient, uint256 amount): Transfers tokens from the caller to the recipient.
        - allowance(address owner, address spender): Returns the remaining number of tokens that the spender can spend on behalf of the owner.
        - approve(address spender, uint256 amount): Sets the allowance for the spender to spend the caller's tokens.
        - transferFrom(address sender, address recipient, uint256 amount): Transfers tokens from sender to recipient using the allowance mechanism.
    - Internal Functions:
        - _transfer(address sender, address recipient, uint256 amount)`: Handles the actual transfer of tokens.

        - _mint(address account, uint256 amount)`: Creates new tokens and assigns them to the specified account.
        - _burn(address account, uint256 amount)`: Destroys tokens from the specified account.
        - _approve(address owner, address spender, uint256 amount)`: Sets the allowance for the spender over the owner's tokens.
        - _beforeTokenTransfer(address from, address to, uint256 amount)`: A hook that can be overridden to execute code before any transfer of tokens.
    - Burnable Token:
        - burn(uint256 amount)`: Allows the caller to destroy their own tokens.
        - burnFrom(address account, uint256 amount)`: Allows the caller to destroy tokens from another account, given that they have a sufficient allowance.
- This code provides a robust ERC20 token implementation with additional burn functionalities, suitable for various token-based applications on the Ethereum blockchain.

Audit scope

Name	**Code Review and Security Analysis Report for LSD Smart Contract**
Platform	**Base Chain Network**
Language	**Solidity**
File	LSD.sol
Smart Contract Code	0xcd1b51b87a8a7137d6421ba5a976225187a26777
Audit Date	June 17th,2024
Audit Result	**Passed**

Code Audit History

0	0	0	0	0	0
Total Findings	Critical	High	Medium	Low	Informational

Severity Definitions

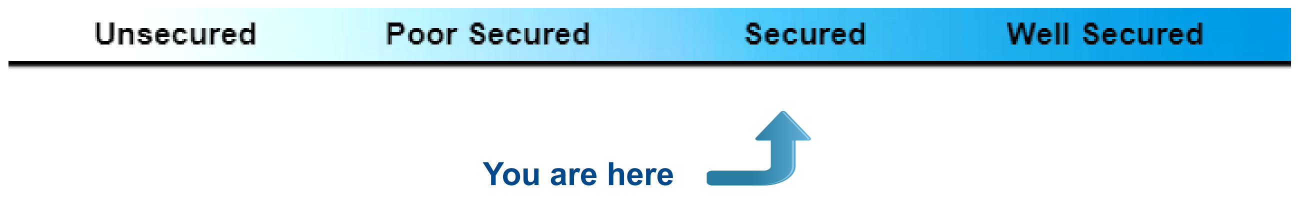
0 🟥	**Critical**	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
0 🟧	**High**	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. Public access is crucial.
0 🟨	**Medium**	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
0 🟩	**Low**	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets, that can't have a significant impact on execution
0 🟦	**Lowest / Informational / Best Practice**	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: L7 Token • Symbol: LSD • Decimals: 18 • Total Supply: 210 million	**YES, This is valid.**
Ownership Control: • There are no owner functions, which makes it 100% decentralized.	**YES, This is valid.**
Other Features: • This code provides a robust ERC20 token implementation with additional burn functionalities, suitable for various token-based applications on the Ethereum blockchain.	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, the Customer`s solidity-based smart contracts are **“Secured”**.This token contract does not have any ownership control, hence it is 100% decentralized.



We used various tools like Slither, Solhint, and Remix IDE. At the same time, this finding is based on a 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. The general overview is presented in the AS-IS section and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium, 0 low, and 0 very low-level issues.

Investor Advice: A 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	The solidity version is not specified	Passed
	The solidity version is too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack 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	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	Passed
	High consumption 'storage' storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage is not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: **PASSED**

Business Risk Analysis

Category	Result
● Buy Tax	0%
● Sell Tax	0%
● Cannot Buy	No
● Cannot Sell	No
● Max Tax	0%
● Modify Tax	No
● Fee Check	Not Detected
● Is Honeypot	Not Detected
● Trading Cooldown	Not Detected
● Can Pause Trade?	Not Detected
● Pause Transfer?	No
● Max Tax?	No
● Is it Anti-whale?	Not Detected
● Is Anti-bot?	Not Detected
● Is it a Blacklist?	No
● Blacklist Check	No
● Can Mint?	No
● Is it a Proxy Contract?	No
● Is it used Open Source?	No
● External Call Risk?	No
● Balance Modifiable?	No
● Can Take Ownership?	No
● Ownership Renounce?	No
● Hidden Owner?	Not Detected
● Self Destruction?	Not Detected
● Auditor Confidence	High

Overall Audit Result: PASSED

Code Quality

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

The libraries in LSD 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 LSD.

The EtherAuthority 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 well commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given an LSD smart contract code in the form of a basescan web link.

As mentioned above, code parts are well commented on. And 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 used in this smart contract infrastructure 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

LSD.sol

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	burn	write	Passed	No Issue
3	burnFrom	write	Passed	No Issue
4	name	read	Passed	No Issue
5	symbol	read	Passed	No Issue
6	decimals	read	Passed	No Issue
7	totalSupply	read	Passed	No Issue
8	balanceOf	read	Passed	No Issue
9	transfer	write	Passed	No Issue
10	allowance	read	Passed	No Issue
11	approve	write	Passed	No Issue
12	transferFrom	write	Passed	No Issue
13	increaseAllowance	write	Passed	No Issue
14	decreaseAllowance	write	Passed	No Issue
15	_transfer	internal	Passed	No Issue
16	_mint	internal	Passed	No Issue
17	_burn	internal	Passed	No Issue
18	_approve	internal	Passed	No Issue
19	_beforeTokenTransfer	internal	Passed	No Issue

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

No Low-severity vulnerabilities were found.

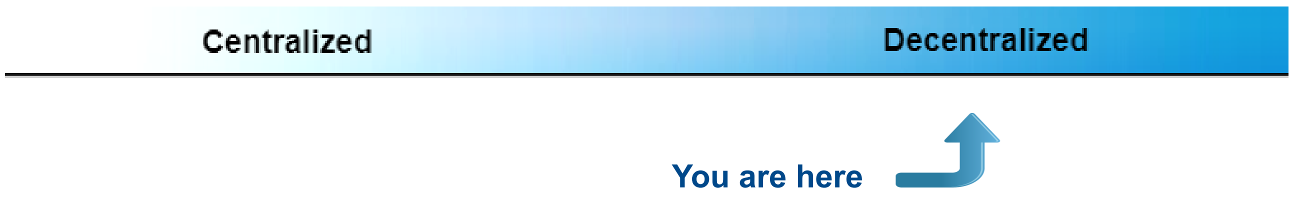
Very Low / Informational / Best practices:

No very-low severity vulnerabilities were found.

Centralization

The LSD smart contract does not have any ownership control, **hence it is 100% decentralized.**

Therefore, there is **no** centralization risk.



Conclusion

We were given a contract code in the form of a basescan web link. And we have used all possible tests based on given objects as files. We observed no issues in the smart contracts. 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 the 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 **"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 the 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 white box penetration testing. We look at the project's website 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, and 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).

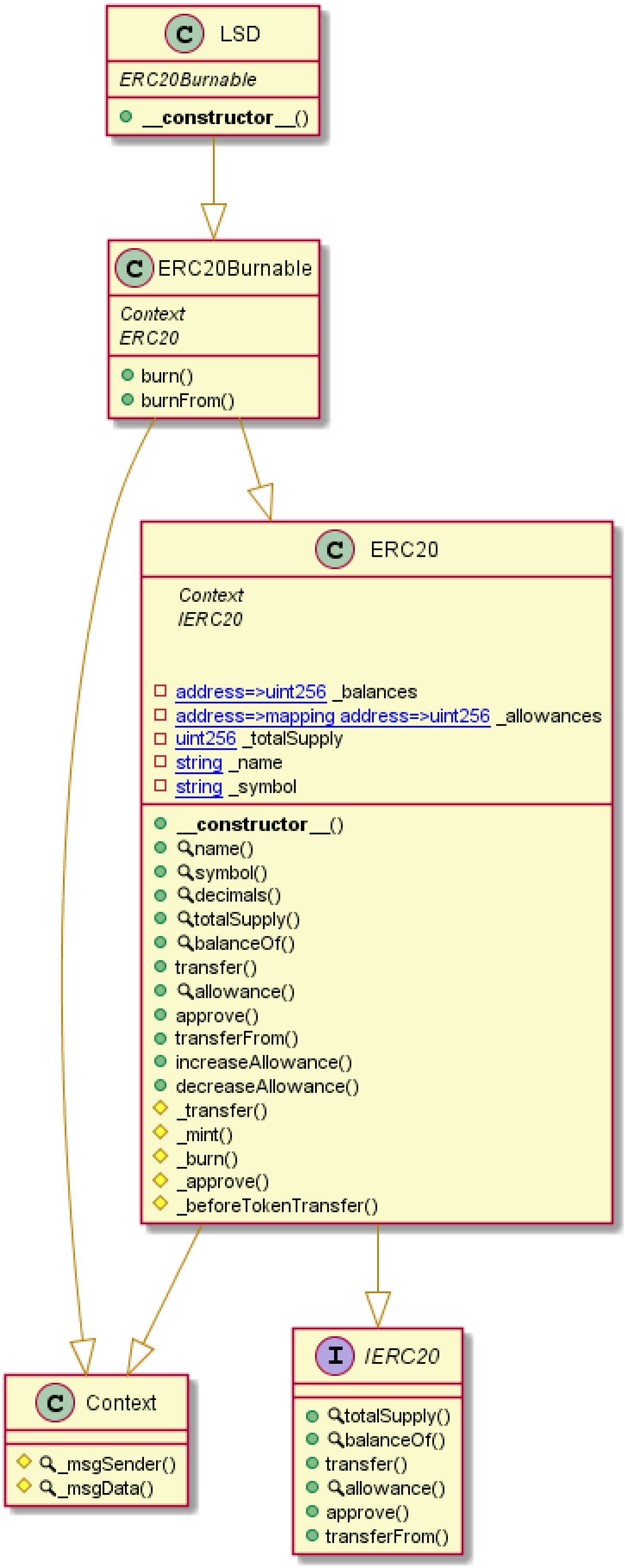
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While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

Appendix

Code Flow Diagram - LSD



Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We did the analysis of the project altogether. Below are the results.

LSD.sol

```
INFO:Detectors:
Context._msgData() (LSD.sol#84-87) 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 (LSD.sol#6) allows old versions
solc-0.8.25 is not recommended for deployment
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Redundant expression "this (LSD.sol#85)" inContext (LSD.sol#79-88)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Slither:LSD.sol analyzed (5 contracts with 93 detectors), 4 result(s) found
```

Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

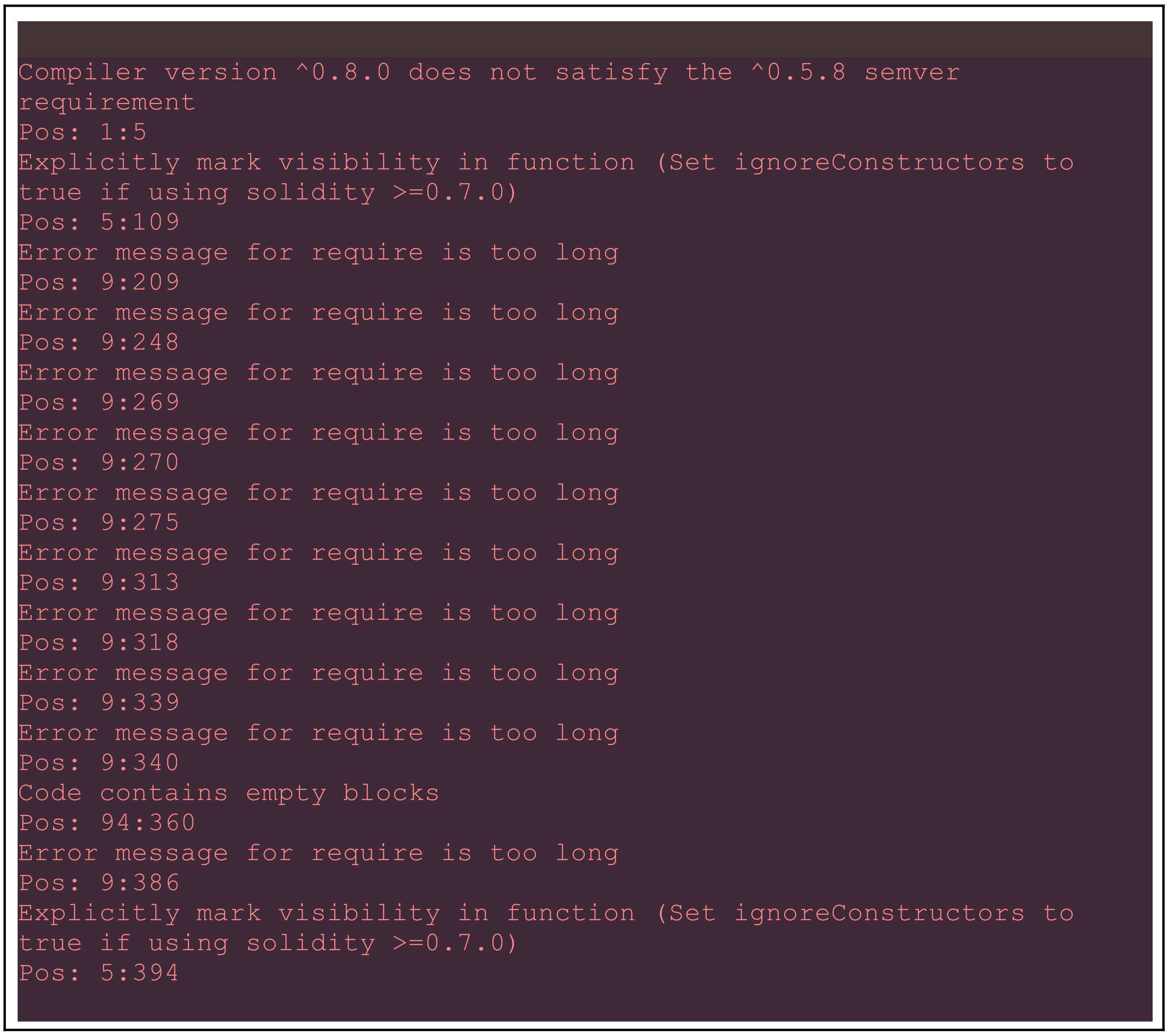
LSD.sol

```
Gas costs:
Gas requirement of function LSD.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: 370:4:

Gas costs:
Gas requirement of function LSD.burnFrom 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: 385:4:

Constant/View/Pure functions:
ERC20._beforeTokenTransfer(address,address,uint256)    :    Potentially    should    be
constant/view/pure but is not.
Pos: 361:4:

Similar variable names:
ERC20._burn(address,uint256) : Variables have very similar names "account" and "amount".
Pos: 314:16:

Similar variable names:
ERC20Burnable.burnFrom(address,uint256) : Variables have very similar names "account" and
"amount".
Pos: 389:23:

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.
Pos: 387:8:
```

Solhint Linter

Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

LSD.sol



```
Compiler version ^0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:5
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:109
Error message for require is too long
Pos: 9:209
Error message for require is too long
Pos: 9:248
Error message for require is too long
Pos: 9:269
Error message for require is too long
Pos: 9:270
Error message for require is too long
Pos: 9:275
Error message for require is too long
Pos: 9:313
Error message for require is too long
Pos: 9:318
Error message for require is too long
Pos: 9:339
Error message for require is too long
Pos: 9:340
Code contains empty blocks
Pos: 94:360
Error message for require is too long
Pos: 9:386
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:394
```

Software analysis result:

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