Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	Wrapped liquid staked Ether
Website:	lido.fi
Platform:	Ethereum
Language:	Solidity
Date:	April 10th, 2025

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

As part of EtherAuthority's community smart contract audit initiatives, the smart contract of WstETH Token from lido.fi was audited. 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 April 10th, 2025.

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 WstETH contract is a wrapper for the StETH token, designed to provide a non-rebasing version of staked ETH for better compatibility with DeFi protocols. Unlike StETH, which rebases as staking rewards and penalties are applied, WstETH maintains a static balance, with value accruing through exchange rate appreciation.

Key Features

- **Wrapping & Unwrapping StETH**
  - Users can wrap StETH into WstETH via the wrap function.
  - Unwrapping WstETH back into StETH is done using the unwrap function.
- **Non-Rebasing Mechanism**
  - The balance of WstETH remains constant, avoiding changes due to staking rewards and penalties.
  - Value is reflected through a changing exchange rate between WstETH and StETH.
- **ETH Staking Shortcut**
  - Users can send ETH directly to the contract to stake it in Lido (stETH.submit()), receiving WstETH in return.
  - This simplifies the staking process by automatically wrapping the received StETH.
- **Conversion Rate Queries**

- getWstETHByStETH(uint256 _stETHAmount): Returns the amount of WstETH for a given amount of StETH.
- getStETHByWstETH(uint256 _wstETHAmount): Returns the amount of StETH for a given amount of WstETH.
- stEthPerToken(): Gets the current amount of StETH per 1 WstETH.
- tokensPerStEth(): Gets the current amount of WstETH per 1 StETH.

Audit scope

Name	**Code Review and Security Analysis Report for Wrapped liquid staked Ether 2.0 Token Smart Contract**
Platform	**Ethereum**
File	WstETH.sol
Smart Contract Code	0x7f39c581f595b53c5cb19bd0b3f8da6c935e2ca0
Audit Date	April 10th, 2025

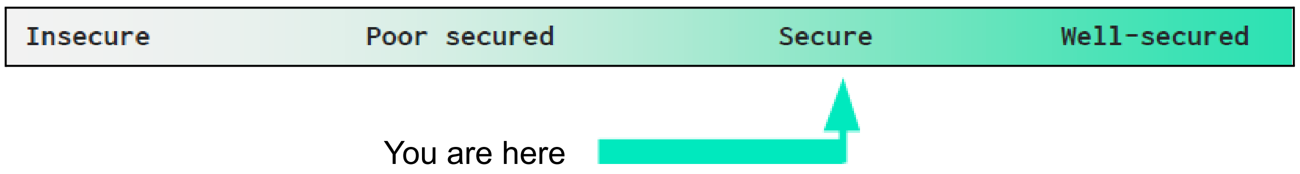
Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: Wrapped liquid staked Ether 2.0 • Symbol: WstETH • Decimals: 1	**YES, This is valid.**
Key Features: **1. Wrapping & Unwrapping Mechanism** • **wrap(uint256 _stETHAmount) → uint256** ○ Converts StETH into WstETH at the current exchange rate. ○ Requires prior approval of StETH transfer. ○ Mints WstETH to the user and transfers StETH to the contract. • **unwrap(uint256 _wstETHAmount) → uint256** ○ Converts WstETH back into StETH. ○ Burns WstETH from the user and transfers the equivalent amount of StETH. **2. Staking ETH Directly** • **receive() external payable** ○ Allows users to send ETH directly to the contract. ○ The contract stakes ETH in Lido (stETH.submit()). ○ Mints the equivalent WstETH to the sender. **3. Conversion Rate Calculations** • **getWstETHByStETH(uint256 _stETHAmount) → uint256**	

○ Returns the amount of WstETH that corresponds to a given StETH amount. ● **getStETHByWstETH(uint256 _wstETHAmount) → uint256** ○ Returns the amount of StETH that corresponds to a given WstETH amount. ● **stEthPerToken() → uint256** ○ Returns how much StETH corresponds to 1 WstETH. ● **tokensPerStEth() → uint256** ○ Returns how much WstETH corresponds to 1 StETH. **4. Token Properties** ● **Static Balance**: Unlike StETH, WstETH does not rebase. Instead, its value appreciates over time. ● **ERC20 Compliance**: Inherits ERC20 and ERC20Permit for standard token functionalities and gasless approvals. ● **Non-Upgradable**: The contract is immutable once deployed. **5. Security Considerations** ● **Requires Approval for Wrapping**: Users must approve StETH transfers before wrapping. ● **No Reentrancy Protection**: This relies on StETH's security, as it interacts with external transfers. ● **No Admin Functions**: Fully decentralized with no owner or privileged roles.	
Ownership Control: ● There are no owner functions, which makes it 100% decentralized.	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, the Customer's solidity-based smart contract is **"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 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, 1 low, and 2 very low-level issues.

Investors' 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	Solidity version not specified	Passed
	Solidity version is 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	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	Not Detected
● Fee Check	No
● Is Honeypot	Not Detected
● Trading Cooldown	Not Detected
● Can Pause Trade?	No
● 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?	No
● Can Take Ownership?	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, inheritance, and Interfaces. This is a compact and well-written smart contract.

The libraries in WstETH 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 WstETH Token.

The EtherAuthority team has no 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 contract. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a WstETH Token smart contract code in the form of an Etherscan 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 and 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

WstETH.sol : Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	wrap	external	Critical operation lacks event log, Lack of an Emergency Pause Mechanism	Refer Audit Findings
3	unwrap	external	Critical operation lacks event log, Lack of an Emergency Pause Mechanism	Refer Audit Findings
4	receive	external	Lack of Emergency Pause Mechanism	Refer Audit Findings
5	getWstETHByStETH	external	Passed	No Issue
6	getStETHByWstETH	external	Passed	No Issue
7	stEthPerToken	external	Passed	No Issue
8	tokensPerStEth	external	Passed	No Issue
9	permit	write	Passed	No Issue
10	nonces	read	Passed	No Issue
11	DOMAIN_SEPARATOR	external	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 a 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 being lost
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets, which can't have a 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) Critical operation lacks event log:

Missing event log for:

- wrap
- unwrap

Recommendation: Please write an event log for the listed events.

Very Low / Informational / Best practices:

(1) Lack of Emergency Pause Mechanism:

There is no way to pause key functions like wrap(), unwrap(), and receive() in case of an emergency or vulnerability discovery.

Recommendation: Implement a pause() function using OpenZeppelin’s Pausable contract to halt operations if necessary.

(2) Use the latest solidity version:

Using the latest solidity will prevent any compiler-level bugs.

Recommendation: We suggest using version > 0.8.0.

Centralization Risk

The WstETH Token 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 Etherscan web links. We have used all possible tests based on the given objects as files. We observed 1 low and 2 informational issues in the smart contract. And those issues are not critical. So, **it's good to go for the production**.

Since possible test cases can be unlimited for such smart contract protocols, 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 the 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 whitebox 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 are 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).

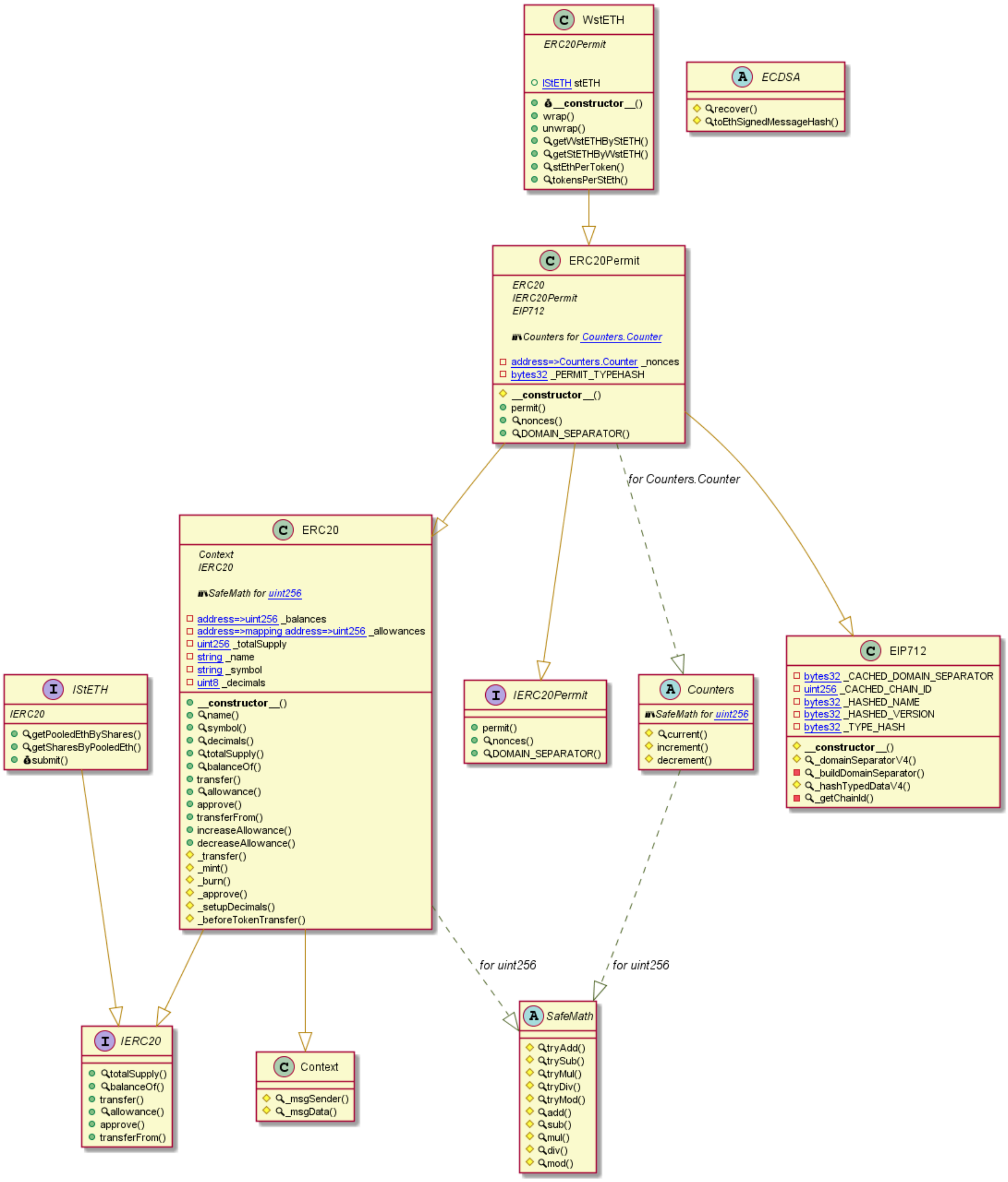
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While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

Appendix

Code Flow Diagram - Wrapped liquid staked Ether 2.0 (WstETH) Token



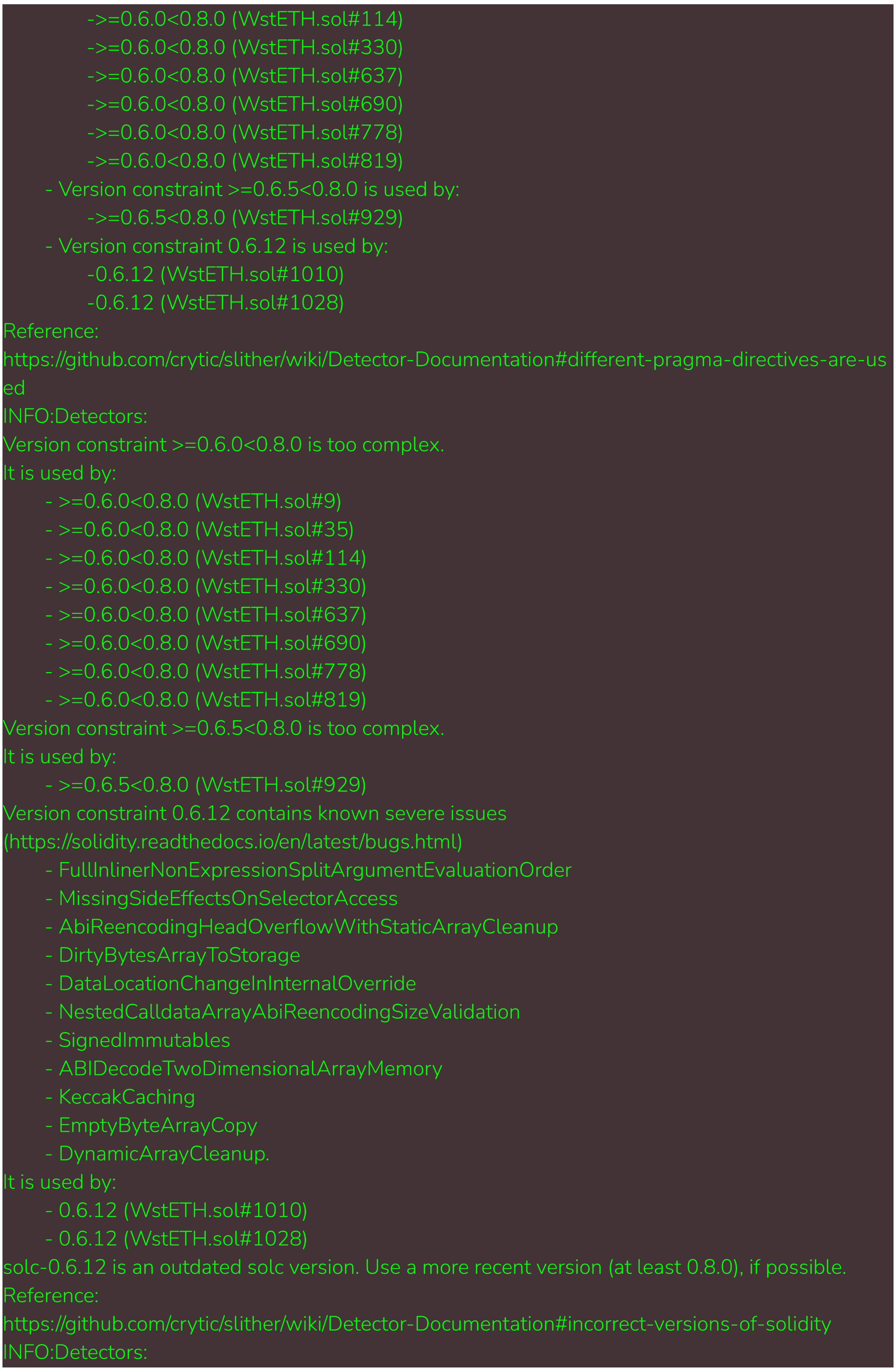
Slither Results Log

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.

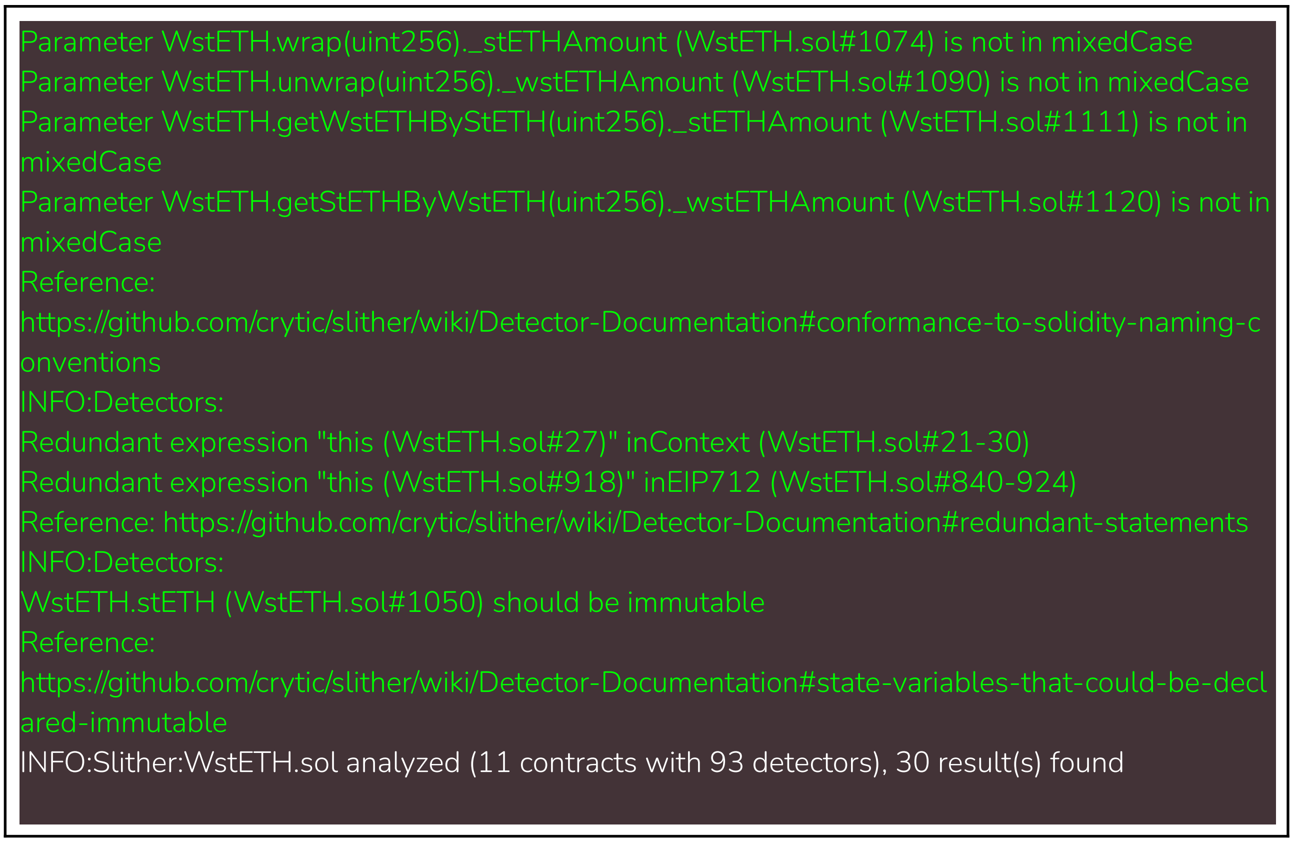
We analyzed the project altogether. Below are the results.

Slither Log >> WstETH.sol

```
INFO:Detectors:
WstETH.wrap(uint256) (WstETH.sol#1074-1080) ignores return value by
stETH.transferFrom(msg.sender,address(this),_stETHAmount) (WstETH.sol#1078)
WstETH.unwrap(uint256) (WstETH.sol#1090-1096) ignores return value by
stETH.transfer(msg.sender,stETHAmount) (WstETH.sol#1094)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
Reentrancy in WstETH.receive() (WstETH.sol#1101-1104):
	External calls:
	- shares = stETH.submit{value: msg.value}(address(0)) (WstETH.sol#1102)
	State variables written after the call(s):
	- _mint(msg.sender,shares) (WstETH.sol#1103)
		- _balances[account] = _balances[account].add(amount) (WstETH.sol#560)
	- _mint(msg.sender,shares) (WstETH.sol#1103)
		- _totalSupply = _totalSupply.add(amount) (WstETH.sol#559)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in WstETH.receive() (WstETH.sol#1101-1104):
	External calls:
	- shares = stETH.submit{value: msg.value}(address(0)) (WstETH.sol#1102)
	Event emitted after the call(s):
	- Transfer(address(0),account,amount) (WstETH.sol#561)
		- _mint(msg.sender,shares) (WstETH.sol#1103)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
3 different versions of Solidity are used:
	- Version constraint >=0.6.0<0.8.0 is used by:
		->=0.6.0<0.8.0 (WstETH.sol#9)
		->=0.6.0<0.8.0 (WstETH.sol#35)
```



```
        ->=0.6.0<0.8.0 (WstETH.sol#114)
        ->=0.6.0<0.8.0 (WstETH.sol#330)
        ->=0.6.0<0.8.0 (WstETH.sol#637)
        ->=0.6.0<0.8.0 (WstETH.sol#690)
        ->=0.6.0<0.8.0 (WstETH.sol#778)
        ->=0.6.0<0.8.0 (WstETH.sol#819)
    - Version constraint >=0.6.5<0.8.0 is used by:
        ->=0.6.5<0.8.0 (WstETH.sol#929)
    - Version constraint 0.6.12 is used by:
        -0.6.12 (WstETH.sol#1010)
        -0.6.12 (WstETH.sol#1028)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-us
ed
INFO:Detectors:
Version constraint >=0.6.0<0.8.0 is too complex.
It is used by:
    - >=0.6.0<0.8.0 (WstETH.sol#9)
    - >=0.6.0<0.8.0 (WstETH.sol#35)
    - >=0.6.0<0.8.0 (WstETH.sol#114)
    - >=0.6.0<0.8.0 (WstETH.sol#330)
    - >=0.6.0<0.8.0 (WstETH.sol#637)
    - >=0.6.0<0.8.0 (WstETH.sol#690)
    - >=0.6.0<0.8.0 (WstETH.sol#778)
    - >=0.6.0<0.8.0 (WstETH.sol#819)
Version constraint >=0.6.5<0.8.0 is too complex.
It is used by:
    - >=0.6.5<0.8.0 (WstETH.sol#929)
Version constraint 0.6.12 contains known severe issues
(https://solidity.readthedocs.io/en/latest/bugs.html)
    - FullInlinerNonExpressionSplitArgumentEvaluationOrder
    - MissingSideEffectsOnSelectorAccess
    - AbiReencodingHeadOverflowWithStaticArrayCleanup
    - DirtyBytesArrayToStorage
    - DataLocationChangeInInternalOverride
    - NestedCalldataArrayAbiReencodingSizeValidation
    - SignedImmutables
    - ABIDecodeTwoDimensionalArrayMemory
    - KeccakCaching
    - EmptyByteArrayCopy
    - DynamicArrayCleanup.
It is used by:
    - 0.6.12 (WstETH.sol#1010)
    - 0.6.12 (WstETH.sol#1028)
solc-0.6.12 is an outdated solc version. Use a more recent version (at least 0.8.0), if possible.
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
```

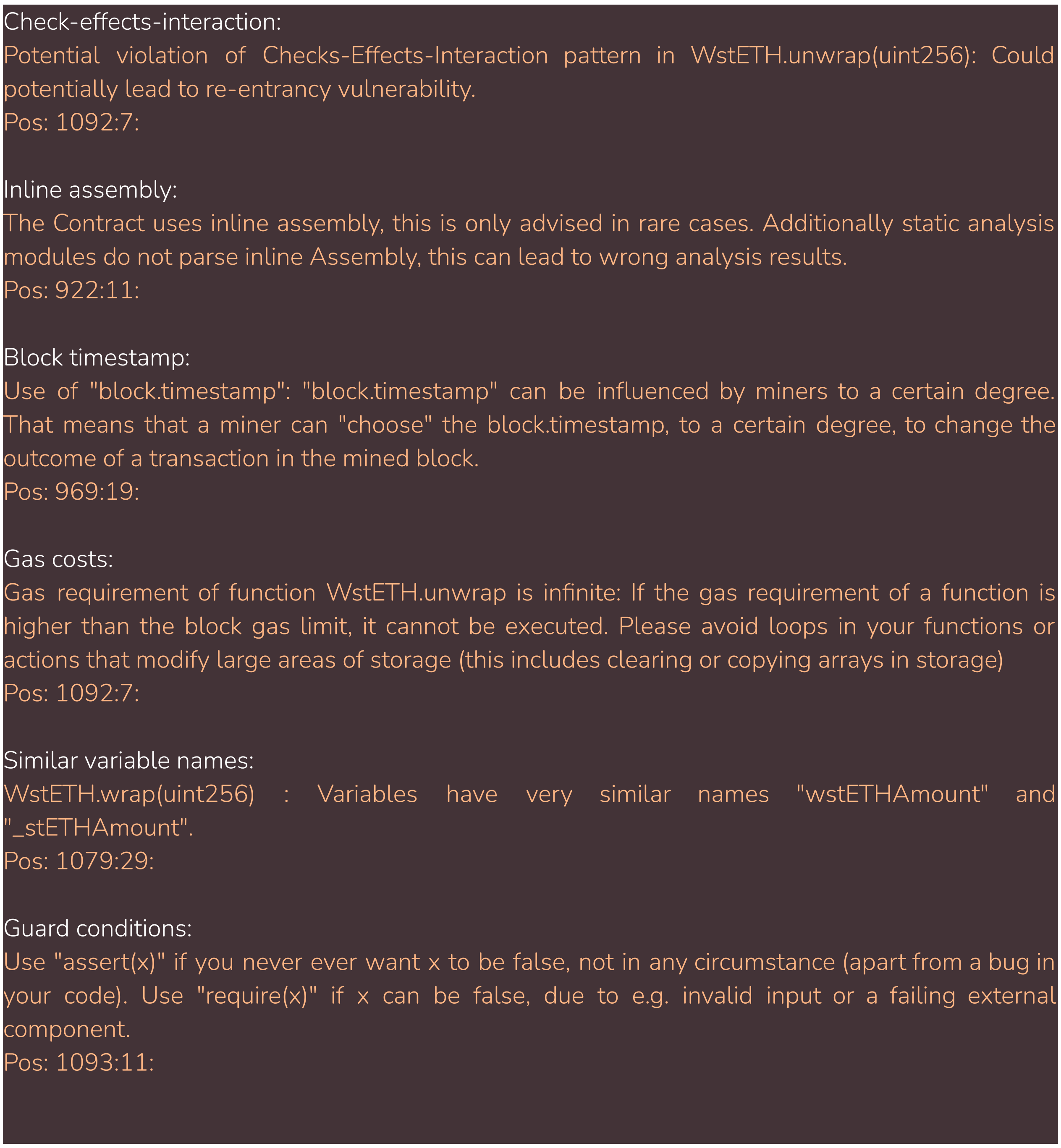


```
Parameter WstETH.wrap(uint256)._stETHAmount (WstETH.sol#1074) is not in mixedCase
Parameter WstETH.unwrap(uint256)._wstETHAmount (WstETH.sol#1090) is not in mixedCase
Parameter WstETH.getWstETHByStETH(uint256)._stETHAmount (WstETH.sol#1111) is not in
mixedCase
Parameter WstETH.getStETHByWstETH(uint256)._wstETHAmount (WstETH.sol#1120) is not in
mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-c
onventions
INFO:Detectors:
Redundant expression "this (WstETH.sol#27)" inContext (WstETH.sol#21-30)
Redundant expression "this (WstETH.sol#918)" inEIP712 (WstETH.sol#840-924)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
WstETH.stETH (WstETH.sol#1050) should be immutable
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-decl
ared-immutable
INFO:Slither:WstETH.sol analyzed (11 contracts with 93 detectors), 30 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.

WstETH.sol

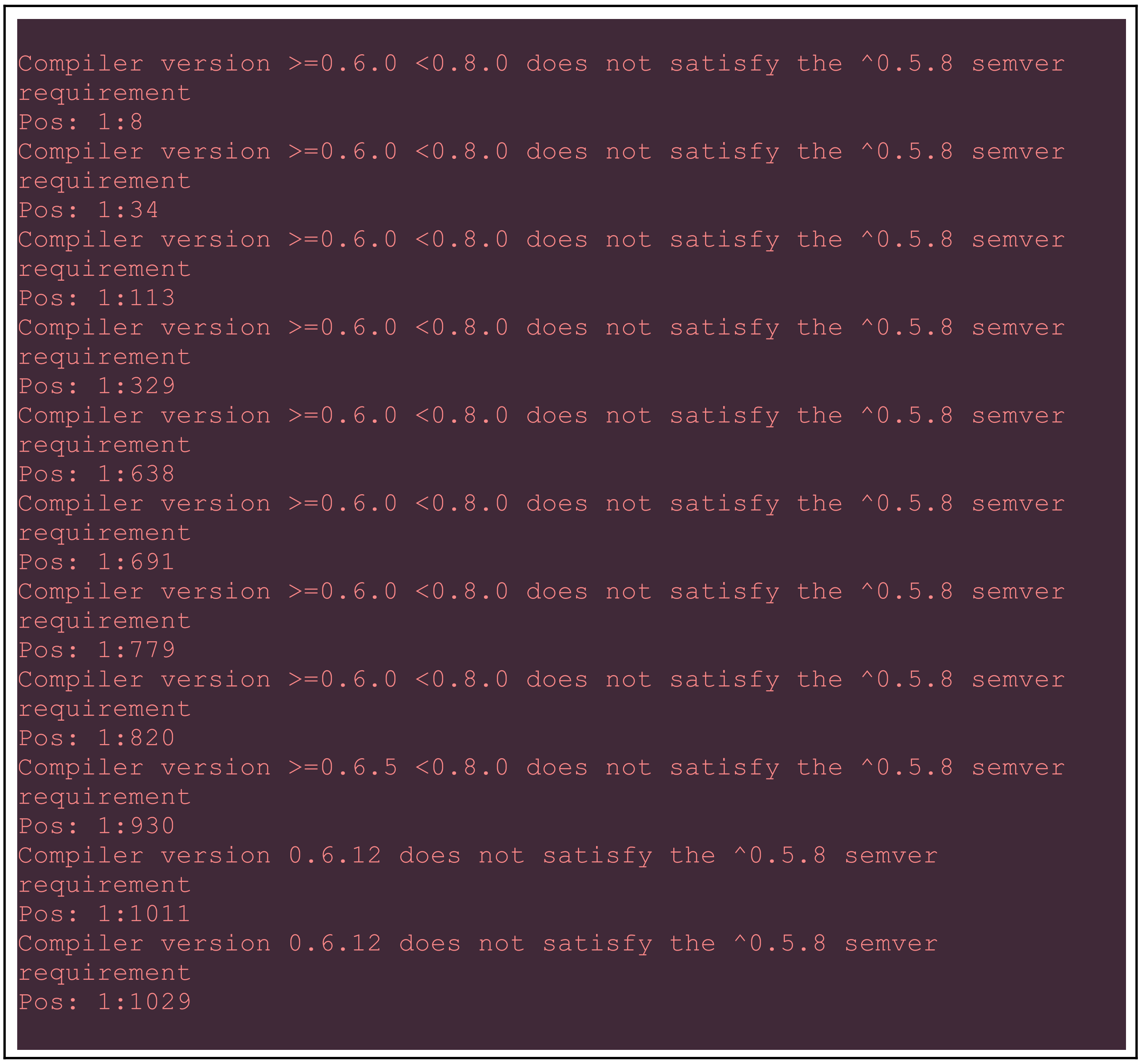


```
Check-effects-interaction:
Potential violation of Checks-Effects-Interaction pattern in WstETH.unwrap(uint256): Could potentially lead to re-entrancy vulnerability.
Pos: 1092:7:

Inline assembly:
The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis modules do not parse inline Assembly, this can lead to wrong analysis results.
Pos: 922:11:

Block timestamp:
Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.
Pos: 969:19:

Gas costs:
Gas requirement of function WstETH.unwrap 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: 1092:7:

Similar variable names:
WstETH.wrap(uint256) : Variables have very similar names "wstETHAmount" and "_stETHAmount".
Pos: 1079:29:

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: 1093:11:
```

Solhint Linter

Solhint 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.

WstETH.sol



```
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:8
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:34
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:113
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:329
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:638
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:691
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:779
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:820
Compiler version >=0.6.5 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:930
Compiler version 0.6.12 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:1011
Compiler version 0.6.12 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:1029
```

Software analysis result:

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