Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	Wrapped SOL (Wormhole) (SOL)
Website:	wormholebridge.com
Platform:	Polygon
Language:	Solidity
Date:	April 7th, 2025

Table of Contents

THIS IS A SECURITY AUDIT REPORT DOCUMENT THAT MAY CONTAIN INFORMATION THAT 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 the SOL Token from wormholebridge.com was audited. The audit was performed using manual analysis and automated software tools. This report presents all the findings regarding the audit performed on April 7th, 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 code implements an **upgradeable proxy pattern** using a BeaconProxy, allowing multiple proxy contracts to share a single logic implementation via a **beacon contract**. This setup separates storage (in the proxy) from logic (in the implementation), and allows upgrades to be made centrally by changing the implementation address stored in the beacon.

- **BeaconProxy Contract**

  - Inherits from OpenZeppelin's Proxy and ERC1967Upgrade.
  - A **beacon contract** is used to determine the implementation address for delegate calls.
  - Stores the beacon address in a storage slot defined by EIP-1967 to avoid collisions with the beacon contract, which is used to implement the state.
  - On deployment, it optionally executes an initialization call (data) on the implementation.

- **BridgeToken Contract**

  - A lightweight contract that **inherits** from BeaconProxy.
  - Its constructor simply passes the beacon address and initialization data to the BeaconProxy constructor.
  - Intended to be used as a token contract that can be **mass-deployed and upgraded** via the beacon.

Audit scope

Name	Code Review and Security Analysis Report for Wrapped SOL Token Smart Contract
Platform	Polygon
File	BridgeToken.sol
Smart Contract	0xd93f7e271cb87c23aaa73edc008a79646d1f9912
Audit Date	April 7th, 2025

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Key Features: • **Beacon Proxy-Based Upgradeability:** ○ BridgeToken inherits from BeaconProxy, making it fully upgradeable. ○ It gets its implementation logic from a beacon contract, not hardcoded in the proxy. • **Custom Initialization Per Instance:** ○ Accepts a byte memory data parameter in the constructor. ○ This data is used for an optional delegatecall to the implementation (used like a constructor for initialization). ○ Example use: initializing token name, symbol, decimals, etc., on deployment. • **Storage Isolation:** ○ Each BridgeToken has its own state (like balances and metadata), even though logic is shared via the beacon. • **Supports Mass Upgradeability:** ○ Because it uses a beacon, upgrading the implementation in the beacon upgrades all deployed BridgeTokens. • **Lightweight Deployment:** ○ The contract has minimal logic—just a pass-through to BeaconProxy—allowing cheap deployments via a factory or bridge system.	

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 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, 1 low, and 4 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 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.	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	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared.	Passed
	Use keywords/functions to be deprecated.	Passed
	Unused code	Moderated
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**

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 SOL 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 SOL 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 SOL Token smart contract code in the form of a polygonscan 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 that is 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

BridgeToken.sol: Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Improper Input Validation in Constructor	Refer Audit Findings
2	_beacon	internal	Passed	No Issue
3	_implementation	internal	Passed	No Issue
4	_setBeacon	internal	Passed	No Issue
5	_getImplementation	internal	Passed	No Issue
6	_setImplementation	write	Passed	No Issue
7	_upgradeTo	internal	Passed	No Issue
8	_upgradeToAndCall	internal	Passed	No Issue
9	_upgradeToAndCallSecure	internal	Passed	No Issue
10	_getAdmin	internal	Passed	No Issue
11	_setAdmin	write	Passed	No Issue
12	_changeAdmin	internal	Passed	No Issue
13	_getBeacon	internal	Passed	No Issue
14	_setBeacon	write	No Access Control on `_upgradeBeaconToAndCall` and `_setBeacon	Refer Audit Findings
15	_upgradeBeaconToAndCall	internal	No Access Control on `_upgradeBeaconToAndCall` and `_setBeacon	Refer Audit Findings
16	_delegate	internal	Passed	No Issue
17	_implementation	internal	Passed	No Issue
18	_fallback	internal	Passed	No Issue
19	fallback	external	Passed	No Issue
20	receive	external	Passed	No Issue
21	receive	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 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) No Access Control on `_upgradeBeaconToAndCall` and `_setBeacon`:

Functions such as `_setBeacon`, `_upgradeBeaconToAndCall`, and `_setImplementation` are `internal`, meaning any inheriting contract (like `BridgeToken`) can freely upgrade the logic contract or beacon without restriction.

Resolution: Ensure any external/inheritable access to upgrades (like via a `changeImplementation()` or `setBeacon()`) is protected using `onlyOwner` or `onlyAdmin` style modifiers.

Very Low / Informational / Best practices:

(1) Improper Input Validation in Constructor:

```
constructor(address beacon, bytes memory data) BeaconProxy(beacon, data) {

}
```

The `BridgeToken` constructor accepts arbitrary `data` to be passed to the logic contract via delegatecall.

Resolution: Validate `data` format or use standard initialization patterns (e.g., encoded `initialize()` calls) and validate them during deployment.

(2) Duplicate Call in `functionCallWithValue`:

```
function functionCallWithValue(
    address target,
    bytes memory data,
    uint256 value
) internal returns (bytes memory) {
    return functionCallWithValue(target, data, value, "Address: low-level call with value failed");
}

function functionCallWithValue(
    address target,
    bytes memory data,
    uint256 value,
    string memory errorMessage
) internal returns (bytes memory) {
    require(address(this).balance >= value, "Address: insufficient balance for call");
    require(isContract(target), "Address: call to non-contract");

    (bool success, bytes memory returndata) = target.call{value: value}(data);
    return verifyCallResult(success, returndata, errorMessage);
}
```

There are two `functionCallWithValue` functions—one without an error message and one with. The shorter one simply calls the longer one with a default error message.

Resolution: Optionally remove the shorter version if not used, to reduce bytecode size slightly.

(3) Hardcoded EIP1967 Slots Without Explanation:

```
// This is the keccak-256 hash of "eip1967.proxy.rollback" subtracted by 1
bytes32 private constant _ROLLBACK_SLOT = 0x4910fdfa16fed3260ed0e7147f7cc6da11a60208b5b9406d12a635614ffd9143;

/**
 * @dev Storage slot with the address of the current implementation.
 * This is the keccak-256 hash of "eip1967.proxy.implementation" subtracted by 1, and is
 * validated in the constructor.
 */
bytes32 internal constant _IMPLEMENTATION_SLOT = 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
```

Storage slot constants like `_IMPLEMENTATION_SLOT` and `_BEACON_SLOT` are directly hardcoded.

Resolution: Add comments explaining the derivation of these slots per EIP-1967 spec or use inline computations for clarity.

(4) Unused Imports or Library Functions:

```
pragma solidity ^0.8.0;


contract BridgeToken is BeaconProxy {
    constructor(address beacon, bytes memory data) BeaconProxy(beacon, data) {    infinite gas 98600 g

    }
}
```

Some functions or structs in `StorageSlot` or `Address` may not be used by the final contracts (`BridgeToken` doesn't directly use many `Address` utilities).

Resolution: Remove unused imports or methods to keep the code lean and readable.

Centralization Risk

The SOL 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 a polygonscan web link. We have used all possible tests based on the given objects as files. We observed 1 low and 4 informational issues in the smart contract, and those issues are not critical. So, **it's good to go for 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 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 not to 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 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 by the best industry practices at the date of this report, about: 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).

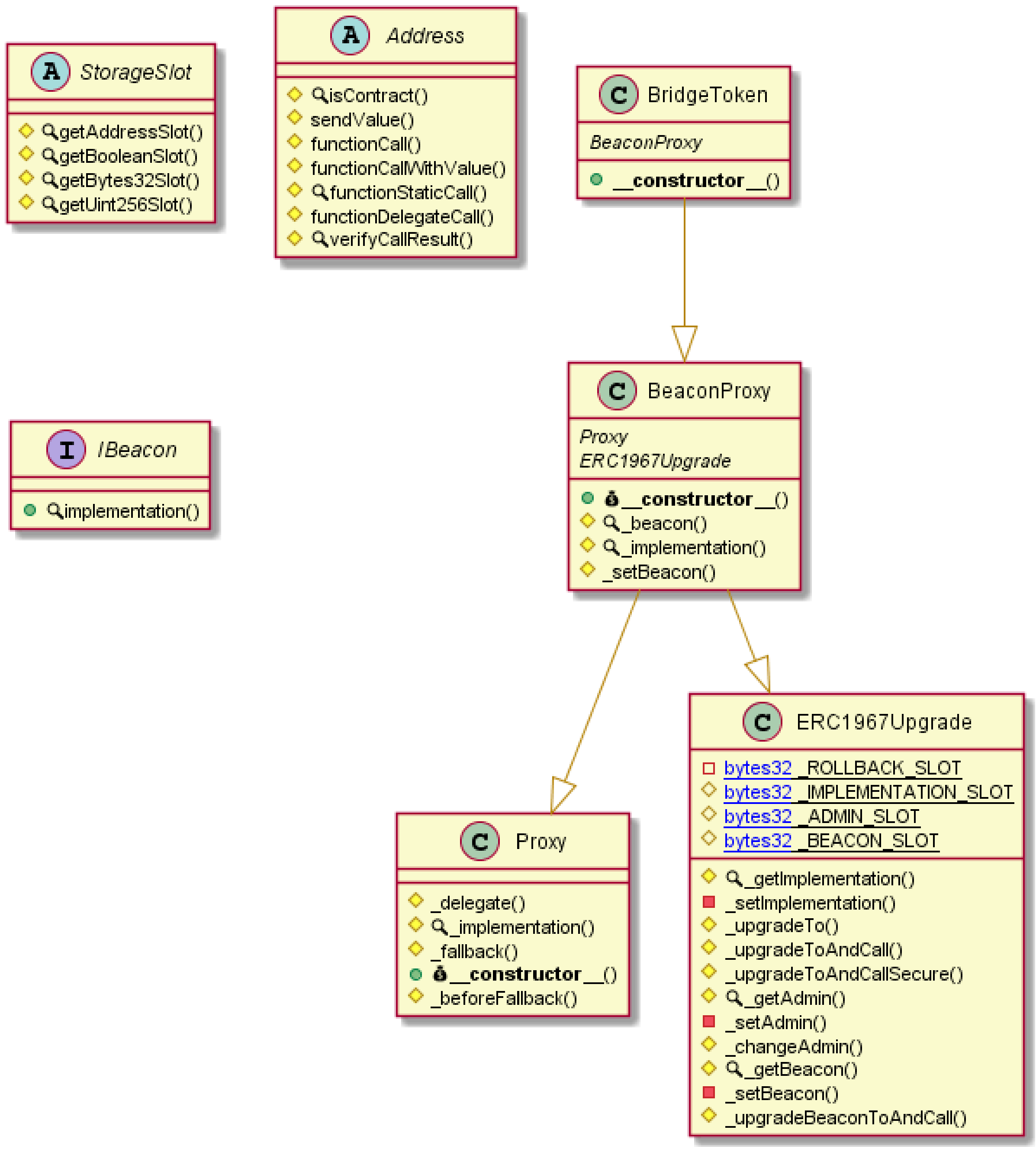
Because the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done 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 SOL 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 did the analysis of the project altogether. Below are the results.

Slither Log >> BridgeToken.sol

```
INFO:Detectors:
Reentrancy in ERC1967Upgrade._upgradeToAndCallSecure(address,bytes,bool)
(BridgeToken.sol#502-530):
	External calls:
	- Address.functionDelegateCall(newImplementation,data) (BridgeToken.sol#512)
	-
Address.functionDelegateCall(newImplementation,abi.encodeWithSignature(upgradeTo(address
),oldImplementation)) (BridgeToken.sol#520-523)
	Event emitted after the call(s):
	- Upgraded(newImplementation) (BridgeToken.sol#478)
		- _upgradeTo(newImplementation) (BridgeToken.sol#528)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Version constraint ^0.8.0 contains known severe issues
(https://solidity.readthedocs.io/en/latest/bugs.html)
	- ^0.8.0 (BridgeToken.sol#12)
	- ^0.8.0 (BridgeToken.sol#99)
	- ^0.8.0 (BridgeToken.sol#319)
	- ^0.8.0 (BridgeToken.sol#408)
	- ^0.8.0 (BridgeToken.sol#623)
	- ^0.8.0 (BridgeToken.sol#687)
Version constraint ^0.8.2 contains known severe issues
(https://solidity.readthedocs.io/en/latest/bugs.html)
	- FullInlinerNonExpressionSplitArgumentEvaluationOrder
	- MissingSideEffectsOnSelectorAccess
	- AbiReencodingHeadOverflowWithStaticArrayCleanup
	- DirtyBytesArrayToStorage
	- DataLocationChangeInInternalOverride
	- NestedCalldataArrayAbiReencodingSizeValidation
	- SignedImmutables
```

```
	- ABIDecodeTwoDimensionalArrayMemory
	- KeccakCaching.
It is used by:
	- ^0.8.2 (BridgeToken.sol#427)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Slither:BridgeToken.sol analyzed (7 contracts with 93 detectors), 29 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.

BridgeToken.sol

```
Low level calls:
Use of "delegatecall": should be avoided whenever possible. External code, that is called can change the state of the calling contract and send ether from the caller's balance. If this is wanted behaviour, use the Solidity library feature if possible.
Pos: 281:50:

Gas costs:
Fallback function of contract BeaconProxy requires too much gas (infinite). If the fallback function requires more than 2300 gas, the contract cannot receive Ether.
Pos: 382:4:

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.
Pos: 649:8:
```


Email: audit@EtherAuthority.io

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.

BridgeToken.sol

```
Compiler version ^0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:11
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 9:87
Compiler version ^0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:98
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 9:127
Avoid to use low level calls.
Pos: 51:280
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 17:302
Compiler version ^0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:318
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 9:337
Code contains empty blocks
Pos: 49:399
Compiler version ^0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:622
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:647
Compiler version ^0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:686
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:690
Code contains empty blocks
Pos: 78:690
```

Software analysis result:

This software reported many false positive results, some of which are informational issues. Therefore, those issues can be safely ignored.

Email: audit@EtherAuthority.io