Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: Wrapped Ton Coin
Website: ton.org
Platform: Ethereum
Language: Solidity
Date: May 12th, 2024

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 contracts audit initiatives, the Wrapped TON Coin smart contracts from ton.org were 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 May 12th, 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

- TON is a decentralized and open network, created by the community using a technology designed by Telegram.
- This Solidity code defines a bridge contract that facilitates the transfer of tokens between the Ethereum and TON (Telegram Open Network) networks. Let's break down the key components and functionalities of the contract:
- Here's a brief overview of the key components and functionalities of the provided code:
    - **Interfaces:** The TonUtils interface defines structs for TON addresses and transactions, as well as a struct for signature data. The IERC20 interface defines the standard ERC20 token functions.
    - **ERC20 Token:** The ERC20 contract implements the standard ERC20 token functionality with functions for transferring tokens, managing allowances, and emitting events.
    - **Bridge Interface:** The BridgeInterface interface extends TonUtils and declares functions for voting on various actions such as minting tokens, updating the set of oracles, and switching burn status.
    - **Signature Checker:** The SignatureChecker contract provides functions for verifying ECDSA signatures and generating unique IDs for different types of actions.
    - **Wrapped TON:** The WrappedTON contract extends ERC20 and TonUtils, adding additional functionalities for minting and burning tokens, specifically for interactions with the TON network.

- **Bridge:** The Bridge contract inherits from SignatureChecker and WrappedTON, implementing the bridge functionality. It maintains a set of oracles, allows for voting on different actions, and executes the actions based on the received votes.

- The contract is without any other custom functionality and without any ownership control, which makes it truly decentralized.
- Overall, the code aims to provide a decentralized bridge between Ethereum and TON networks, allowing for token swaps and governance through a voting mechanism involving a set of oracles.

Audit scope

Name	Code Review and Security Analysis Report for Ton Coin Smart Contract
Platform	**Ethereum**
File	Bridge.sol
Smart Contract Code	0x582d872a1b094fc48f5de31d3b73f2d9be47def1
Audit Date	May 12th, 2024

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: Wrapped TON Coin • Symbol: TONCOIN • Decimals: 9	**YES, This is valid.**
Ownership Control: • There are no owner functions, which makes it 100% decentralized. • Oracles are part of several transactions allowing execution of some functions in a decentralized way.	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, 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 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 3 very low level issues.

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	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	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	Passed
Gas Optimization	"Out of Gas" Issue	Moderated
	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

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?	Not Detected
Max Tax?	No
Is it Anti-whale?	Not Detected
Is Anti-bot?	Not Detected
Is it a Blacklist?	Not Detected
Blacklist Check	No
Can Mint?	Yes
Is it Proxy?	No
Can Take Ownership?	Not Detected
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 contract contains Libraries, Smart contracts, inherits and Interfaces. This is a compact and well written smart contract.

The libraries in Ton Coin 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 contract in the Ton Coin.

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 not well commented on in the smart contracts. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a Ton Coin smart contract code in the form of an Etherscan web link.

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

Bridge Contract: Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	generalVote	internal	Infinite loops, Out of Gas issue	Refer Audit Findings
3	voteForMinting	write	Function input parameters lack of check	Refer Audit Findings
4	voteForNewOracleSet	write	Function input parameters lack of check	Refer Audit Findings
5	voteForSwitchBurn	write	Passed	No Issue
6	executeMinting	internal	Passed	No Issue
7	updateOracleSet	internal	Function input parameters lack of check, Infinite loops, Out of Gas issue	Refer Audit Findings
8	getFullOracleSet	read	Passed	No Issue
9	checkSignature	write	Passed	No Issue
10	getSwapDataId	write	Passed	No Issue
11	getNewSetId	write	Passed	No Issue
12	getNewBurnStatusId	write	Passed	No Issue
13	mint	internal	Passed	No Issue
14	burn	external	Passed	No Issue
15	burnFrom	external	Passed	No Issue
16	decimals	write	Passed	No Issue
17	checkSignature	write	Passed	No Issue
18	getSwapDataId	write	Passed	No Issue
19	getNewSetId	write	Passed	No Issue
20	getNewBurnStatusId	write	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) Function input parameters lack of check:

```
 function updateOracleSet(int oracleSetHash, address[] memory
newSet) internal {
      uint oldSetLen = oraclesSet.length;
      for(uint i = 0; i < oldSetLen; i++) {
        isOracle[oraclesSet[i]] = false;
      }
      oraclesSet = newSet;
      uint newSetLen = oraclesSet.length;
      for(uint i = 0; i < newSetLen; i++) {
        require(!isOracle[newSet[i]], "Duplicate oracle in Set");
        isOracle[newSet[i]] = true;
      }
      emit NewOracleSet(oracleSetHash, newSet);
    }

function voteForNewOracleSet(int oracleSetHash, address[] memory
newOracles, Signature[] memory signatures) override  public {
      bytes32 _id = getNewSetId(oracleSetHash, newOracles);
      require(newOracles.length > 2, "New set is too short");
      generalVote(_id, signatures);
      updateOracleSet(oracleSetHash, newOracles);
```

```
    }

function voteForMinting(SwapData memory data, Signature[] memory
signatures) override public {
      bytes32 _id = getSwapDataId(data);
      generalVote(_id, signatures);
      executeMinting(data);
    }
```

In functions like voteForNewOracleSet, voteForSwitchBurn, and updateOracleSet, ensure that input parameters are properly validated to prevent unexpected behavior or manipulation.

Resolution: We suggest using validation, like for numerical variables that should be greater than 0 and for address-type check variables that are not address(0). For percentage-type variables, values should have some range, like a minimum of 0 and a maximum of 100.

(2) Infinite loops, Out of Gas issue:

```
 function updateOracleSet(int oracleSetHash, address[] memory
newSet) internal {
      uint oldSetLen = oraclesSet.length;
      for(uint i = 0; i < oldSetLen; i++) {
        isOracle[oraclesSet[i]] = false;
      }
      oraclesSet = newSet;
      uint newSetLen = oraclesSet.length;
      for(uint i = 0; i < newSetLen; i++) {
        require(!isOracle[newSet[i]], "Duplicate oracle in Set");
        isOracle[newSet[i]] = true;
      }
      emit NewOracleSet(oracleSetHash, newSet);
    }

function generalVote(bytes32 digest, Signature[] memory signatures)
internal {
      require(signatures.length >= 2 * oraclesSet.length / 3, "Not
```

```
enough signatures");
      require(!finishedVotings[digest], "Vote is already finished");
      uint signum = signatures.length;
      uint last_signer = 0;
      for(uint i=0; i<signum; i++) {
        address signer = signatures[i].signer;
        require(isOracle[signer], "Unauthorized signer");
        uint next_signer = uint(signer);
        require(next_signer > last_signer, "Signatures are not
sorted");
        last_signer = next_signer;
        checkSignature(digest, signatures[i]);
      }
      finishedVotings[digest] = true;
    }
```

As array elements will increase, then it will cost more and more gas. And eventually, it will stop all the functionality. After several hundreds of transactions, all those functions depending on it will stop. We suggest avoiding loops. For example, use mapping to store the array index. And query that data directly, instead of looping through all the elements to find an element.

Resolution: Adjust logic to replace loops with mapping or other code structure.

- generalVote() - signatures.length
- updateOracleSet() - oraclesSet.length

Very Low / Informational / Best practices:

(1) Potential Gas Limit Issues:

As array elements will increase, then it will cost more and more gas. And eventually, it will stop all the functionality. After several hundreds of transactions, all those functions depending on it will stop.

Resolution: Depending on the size of the oracle set and the number of signatures required, the gas cost of executing functions like generalVote could become prohibitive. Ensure that gas limits are not exceeded, especially in loops and complex operations.

(2) Use latest solidity version:

```
pragma solidity ^0.7.0;
```

Use the latest solidity version while contract deployment to prevent any compiler version level bugs.

Resolution: Please use versions greater than 0.8.7.

(3) Missing SPDX license identifier:

```
Bridge.sol: Warning: SPDX license identifier not provided in source file.
Before publishing, consider adding a comment containing
"SPDX-License-Identifier: <SPDX-License>" to each source file. Use
"SPDX-License-Identifier: UNLICENSED" for non-open-source code. Please see
https://spdx.org for more information.
```

Solidity’s new specification requires a valid SPDX license identifier to be included in every smart contract file.

Resolution: Please add a comment for appropriate SPDX license identifier.

Centralization Risk

The Ton Coin 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. And we have used all possible tests based on given objects as files. We had observed 2 low and 3 informational issues in the smart contracts. 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 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 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 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.

Email: audit@EtherAuthority.io

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).

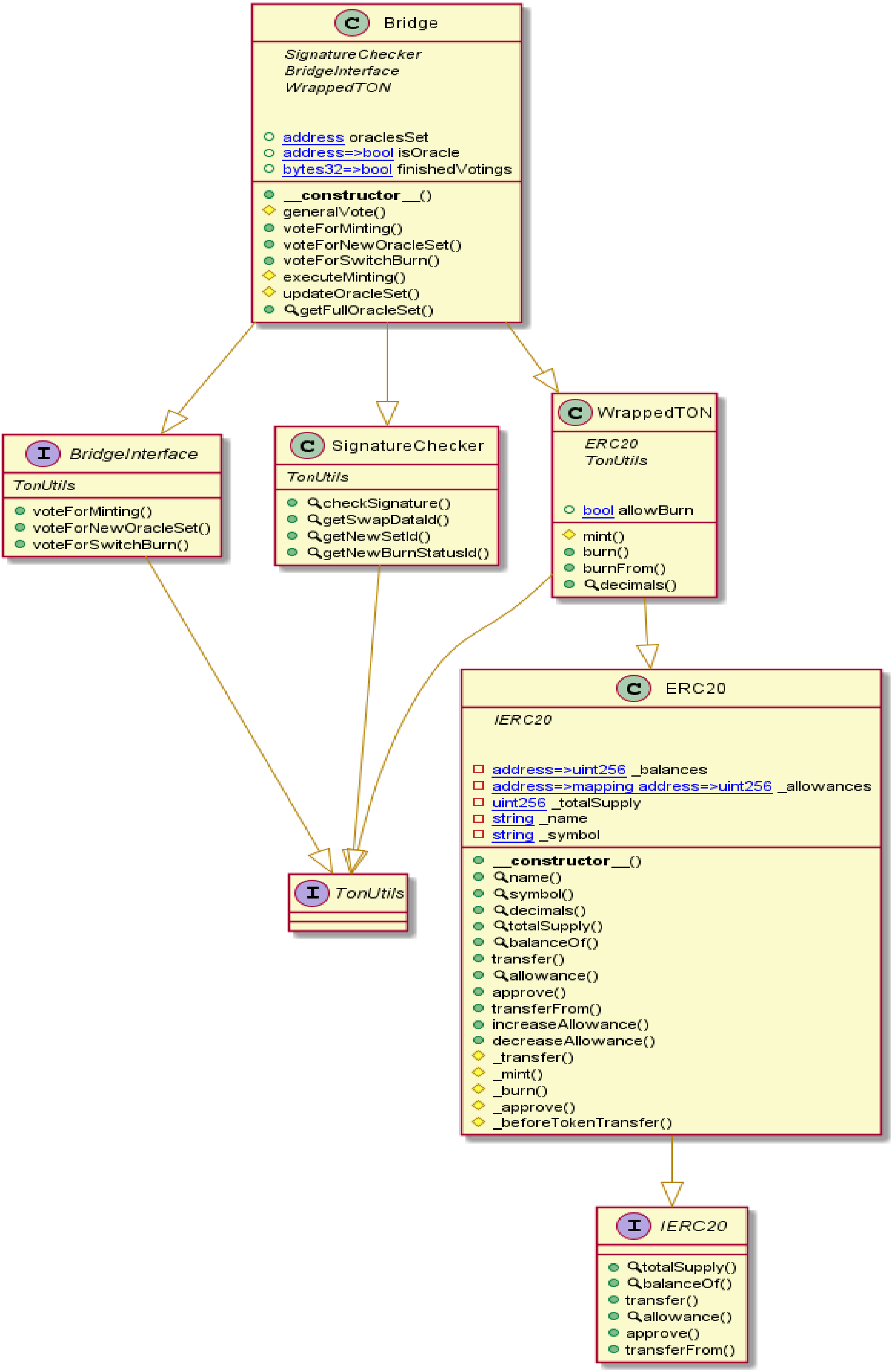
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we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



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 >> Bridge.sol

```
SignatureChecker.checkSignature(bytes32,TonUtils.Signature) (Bridge.sol#379-413) uses assembly
        - INLINE ASM (Bridge.sol#394-398)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

Pragma version^0.7.0 (Bridge.sol#1) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
Bridge.sol analyzed (7 contracts with 84 detectors), 2 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.

Bridge.sol

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.
more
Pos: 397:10:

Gas costs:

Gas requirement of function Bridge.getFullOracleSet 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: 578:4:

Constant/View/Pure functions:

SignatureChecker.getNewBurnStatusId(bool,int256) : Is constant but potentially should not be.
more
Pos: 452:4:

Similar variable names:

Bridge.generalVote(bytes32,struct TonUtils.Signature[]) : Variables have very similar names "signum" and "signer".
Pos: 529:6:

No return:

IERC20.transferFrom(address,address,uint256): Defines a return type but never explicitly returns a value.
Pos: 84:4:

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: 573:8:

Data truncated:

Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.
Pos: 527:35:

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.

Bridge.sol

```
Compiler version ^0.7.6 does not satisfy the ^0.5.8 semver
requirement
Pos: 2:3
Variable name must be in mixedCase
Pos: 9:9
Variable name must be in mixedCase
Pos: 9:13
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:118
Error message for require is too long
Pos: 9:218
Error message for require is too long
Pos: 9:257
Error message for require is too long
Pos: 9:278
Error message for require is too long
Pos: 9:279
Error message for require is too long
Pos: 9:284
Error message for require is too long
Pos: 9:322
Error message for require is too long
Pos: 9:327
Error message for require is too long
Pos: 9:348
Error message for require is too long
Pos: 9:349
Code contains empty blocks
Pos: 94:369
Error message for revert is too long
Pos: 15:406
Error message for revert is too long
Pos: 15:410
Error message for require is too long
Pos: 9:502
Variable name must be in mixedCase
Pos: 46:512
Variable name must be in mixedCase
Pos: 58:512
Variable name must be in mixedCase
Pos: 40:513
Variable name must be in mixedCase
```

```
Pos: 74:513
Explicitly mark visibility in function (Set ignoreConstructors to
true if using solidity >=0.7.0)
Pos: 5:521
Variable name must be in mixedCase
Pos: 7:529
Variable name must be in mixedCase
Pos: 9:533
```

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

Email: audit@EtherAuthority.io