EtherAuthority

etherauthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: SubQueryToken
Website: subquery.network
Platform: Base Chain Network
Language: Solidity
Date: June 25th, 2024

Table of contents

Email: audit@EtherAuthority.io

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the SubQueryToken smart contract from subquery.network 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 25th, 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

- SubQuery Network is a platform designed to simplify the process of querying blockchain data. It provides tools for developers to create, share, and monetize open APIs called SubQuery Projects. These projects can extract, transform, and serve blockchain data in a format that's easy to consume for decentralized applications (dApps) and other blockchain-based projects.
- SubQuery Network simplifies blockchain data access and integration, enabling developers to build powerful applications with ease. By providing robust tooling, infrastructure, and monetization opportunities, SubQuery Network supports a diverse range of use cases across the blockchain ecosystem.

Code Details

- The Solidity code provided defines a set of smart contracts for an ERC20 token called `L2SQToken` that follows the Layer 2 (L2) standard for bridging tokens between Layer 1 (L1) and Layer 2 (L2) networks. The contracts also implement the ERC165 standard for interface detection.
- The `L2SQToken` contract is an example implementation of the L2 standard ERC20 token, named "SubQueryToken" with the symbol "SQT". This token can be minted and burned by the L2 bridge, allowing seamless transfer of tokens between Layer 1 and Layer 2 networks.
- This set of contracts leverages the ERC20 and ERC165 standards to create a token that is compatible with Layer 2 solutions, enabling efficient and scalable token operations across different layers of the blockchain. The `L2SQToken` contract is a concrete example of how to implement a token that can interact with an L2 bridge for minting and burning operations.

Audit scope

Name	**Code Review and Security Analysis Report for SubQueryToken Smart Contract**
Platform	**Base Chain Network**
Language	**Solidity**
File	L2SQToken.sol
Smart Contract Code	0x858c50C3AF1913b0E849aFDB74617388a1a5340d
Audit Date	June 25th,2024
Audit Result	**Passed**

Code Audit History

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

Severity Definitions

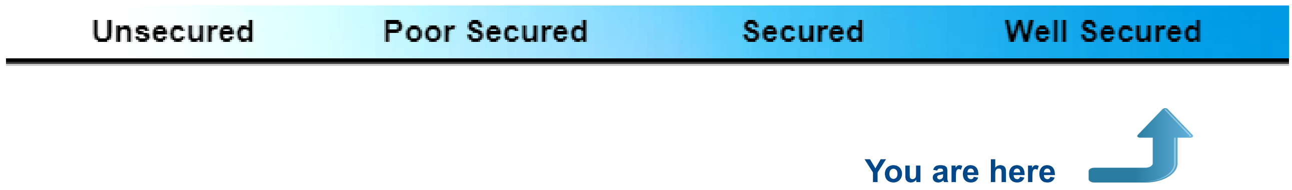
0	**Critical**	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
0	**High**	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. Public access is crucial.
0	**Medium**	Medium-level 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: SubQueryToken • Symbol: SQT • Decimals: 18	YES, This is valid.
L2 Bridge Specifications: • L2 Bridge can mint and burn tokens.	YES, This is valid.
Key Functionalities: • ERC20 Token Operations: The `ERC20` contract implements basic ERC20 functionalities like transfer, approve, and transferFrom, while `_mint` and `_burn` functions enable creating and destroying tokens. • Layer 2 Standard ERC20: The 'L2StandardERC20' contract introduces L2-specific functions'mint' and 'burn', which are only called by the L2 bridge contract and emit 'Mint' and 'Burn' events. • Interface Detection: The `supportsInterface` function enables the contract to specify the interfaces it implements, thereby facilitating compatibility checks.	YES, This is valid.

Audit Summary

According to the standard audit assessment, the Customer`s solidity-based smart contracts are **"Well 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?	Yes
Is it a Proxy Contract?	No
Is it used Open Source?	Yes
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 SubQueryToken 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 SubQueryToken.

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 a SubQueryToken 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

L2SQToken.sol

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	onlyL2Bridge	modifier	Passed	No Issue
3	supportsInterface	write	Passed	No Issue
4	mint	write	access-only L2 Bridge	No Issue
5	burn	write	access-only L2 Bridge	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.

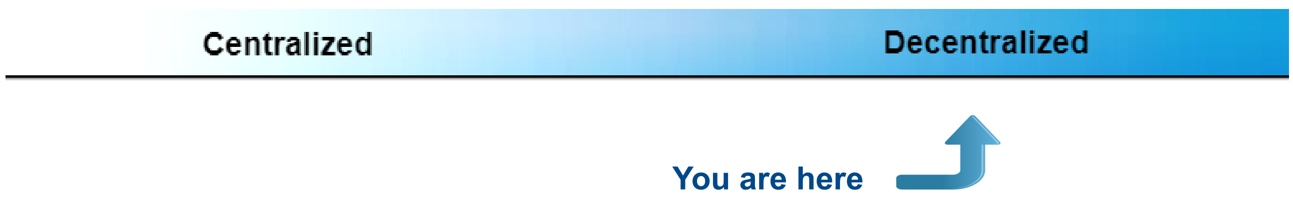
Very Low / Informational / Best practices:

No Very-Low-severity vulnerabilities were found.

Centralization

The SubQueryToken (SQT) 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. We have used all possible tests based on given objects as files. We observed no issue 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 **"Well 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).

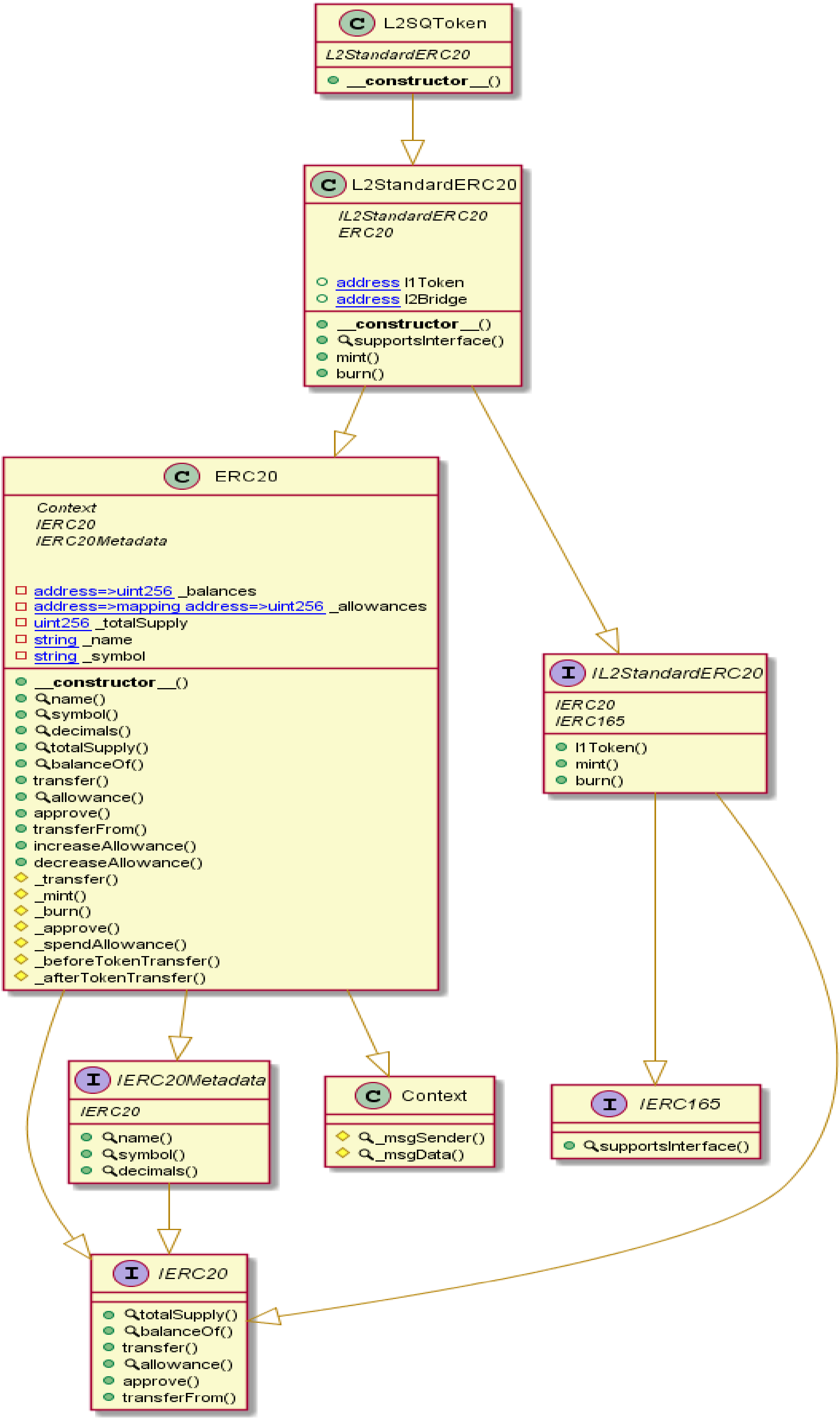
Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While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



Email: audit@EtherAuthority.io

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.

L2SQToken.sol

```
INFO:Detectors:
L2StandardERC20.constructor(address,address,string,string)._name (L2SQToken.sol#474)
shadows:
	- ERC20._name (L2SQToken.sol#127) (state variable)
L2StandardERC20.constructor(address,address,string,string)._symbol (L2SQToken.sol#475)
shadows:
	- ERC20._symbol (L2SQToken.sol#128) (state variable)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
L2StandardERC20.constructor(address,address,string,string)._l1Token (L2SQToken.sol#473)
lacks a zero-check on :
		- l1Token = _l1Token (L2SQToken.sol#477)
L2StandardERC20.constructor(address,address,string,string)._l2Bridge (L2SQToken.sol#472)
lacks a zero-check on :
		- l2Bridge = _l2Bridge (L2SQToken.sol#478)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Context._msgData() (L2SQToken.sol#115-117) 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 (L2SQToken.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:
Parameter L2StandardERC20.supportsInterface(bytes4)._interfaceId (L2SQToken.sol#487) is not
in mixedCase
Parameter L2StandardERC20.mint(address,uint256)._to (L2SQToken.sol#496) is not in
```

```
mixedCase
Parameter L2StandardERC20.mint(address,uint256)._amount (L2SQToken.sol#496) is not in
mixedCase
Parameter L2StandardERC20.burn(address,uint256)._from (L2SQToken.sol#503) is not in
mixedCase
Parameter L2StandardERC20.burn(address,uint256)._amount (L2SQToken.sol#503) is not in
mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-c
onventions
INFO:Detectors:
L2StandardERC20.l1Token (L2SQToken.sol#462) should be immutable
L2StandardERC20.l2Bridge (L2SQToken.sol#463) should be immutable
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-decl
ared-immutable
INFO:Slither:L2SQToken.sol analyzed (8 contracts with 93 detectors), 14 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.

L2SQToken.sol

```
Constant/View/Pure functions:
IL2StandardERC20.burn(address,uint256) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.Note: Import aliases are currently not supported by this static analysis.
Pos: 44:4:

No return:
IERC20Metadata.decimals(): Defines a return type but never explicitly returns a value.
Pos: 24:4:

No return:
IL2StandardERC20.l1Token(): Defines a return type but never explicitly returns a value.
Pos: 40:4:
```

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.

L2SQToken.sol

```
Import statements must be on top
Pos: 1:34
Import statements must be on top
Pos: 1:49
Compiler version ^0.8.0 does not satisfy the ^0.5.8 semver requirement
Pos: 1:5
Use double quotes for string literals
Pos: 33:49
Explicitly mark visibility in function (Set ignoreConstructors to true if using solidity >=0.7.0)
Pos: 5:54
Code contains empty blocks
Pos: 68:57
Use double quotes for string literals
Pos: 44:57
Use double quotes for string literals
Pos: 61:57
```

Software analysis result:

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

Email: audit@EtherAuthority.io