QuillAudits

Audit Report
April, 2021

Burst

Contents

Introduction

This audit report highlights the overall security of the Burst-token contract with commit hash 33b0e2. With this report, we have tried to ensure the reliability of the smart contract by completing the assessment of their system's architecture and smart contract codebase.

Auditing Approach and Methodologies applied

In this audit, I consider the following crucial features of the code.

- Whether the implementation of ERC 20 standards.
- Whether the code is secure.
- Gas Optimization
- Whether the code meets the best coding practices.
- Whether the code meets the SWC Registry issue.

The audit has been performed according to the following procedure:

Manual Audit

- Inspecting the code line by line and revert the initial algorithms of the protocol and then
- compare them with the specification
- Manually analyzing the code for security vulnerabilities.
- Assessing the overall project structure, complexity & quality.
- Checking SWC Registry issues in the code.
- Unit testing by writing custom unit testing for each function.
- Checking whether all the libraries used in the code of the latest version.
- Analysis of security on-chain data.
- Analysis of the failure preparations to check how the smart contract performs in case of bugs and vulnerability.

Automated analysis

- Scanning the project's code base with Mythril, Slither, Echidna, Manticore, others.
- Manually verifying (reject or confirm) all the issues found by tools.
- Performing Unit testing.
- Manual Security Testing (SWC-Registry, Overflow)
- Running the tests and checking their coverage.

Audit Details

Project Name: BURST Token
Token symbol: BURST

Languages: Solidity
Platforms and Tools: Remix, VScode, slither and other tools mentioned in the automated analysis section.

Audit Goals

The focus of the audit was to verify that the Smart Contract System is secure, resilient and working according to the specifications. The audit activities can be grouped in the following three categories:

Security
Identifying security related issues within each contract and the system of contract.

Sound Architecture
Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality
A full review of the contract source code. The primary areas of focus include:

- Accuracy
- Readability
- Sections of code with high complexity

Issue Categories

Every issue in this report was assigned a severity level from the following:

High severity issues

Issues on this level are critical to the smart contract's performance/ functionality and should be fixed before moving to a live environment.

Medium severity issues

Issues on this level could potentially bring problems and should eventually be fixed.

Low severity issues

Issues on this level are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Number of issues per severity

	High	Medium	Low	Suggestions
Open	0	0	5	2

Manual Audit

SWC Registry test

We have tested some known SWC registry issues. Out of all tests only SWC 102 and 103. Both are low priority. We have about it above already.

Serial No.	Description	Comments
SWC-132	Unexpected Ether balance	Pass: Avoided strict equality checks for the Ether balance in a contract
SWC-131	Presence of unused variables	Pass: No unused variables
SWC-128	DoS With Block Gas Limit	Pass
SWC-122	Lack of Proper Signature Verification	Pass
SWC-120	Weak Sources of Randomness from Chain Attributes	Found
SWC-119	Shadowing State Variables	Pass: No ambiguous found.
SWC-118	Incorrect Constructor Name	Pass. No incorrect constructor name used
SWC-116	Timestamp Dependence	Pass: No random value used insufficiently
SWC-115	Authorization through tx.origin	Pass: No tx.origin found
SWC-114	Transaction Order Dependence	Pass

Serial No.	Description	Comments
SWC-113	DoS with Failed Call	Pass: No failed call
SWC-112	Delegatecall to Untrusted Callee	Pass
SWC-111	Use of Deprecated Solidity Functions	Pass : No deprecated function used
SWC-108	State Variable Default Visibility	Pass: Explicitly defined visibility for all state variables
SWC-107	Reentrancy	Pass: Properly used
SWC-106	Unprotected SELF-DESTRUCT Instruction	Pass: Not found any such vulnerability
SWC-104	Unchecked Call Return Value	Pass: Not found any such vulnerability
SWC-103	Floating Pragma	Found
SWC-102	Outdated Compiler Version	Found: Latest version is Version 0.8.3. In code 0.5.0 is used
SWC-101	Integer Overflow and Underflow	Found

High level severity issues

No issues found

Medium level severity issues

No issues found

Low level severity issues

1. **Description** → **SWC 102**: Outdated Compiler Version

```
pragma solidity ^0.5.0;

// ----------------------------------------------------------------------------
// ERC Token Standard #20 Interface
//
// ----------------------------------------------------------------------------
```

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Remediation
It is recommended to use a recent version of the Solidity compiler which is Version 0.8.3.

2. **Description** → **SWC 103: Floating Pragma** [Line 1]

```
pragma solidity ^0.5.0;

// ----------------------------------------------------------------------------
// ERC Token Standard #20 Interface
//
// ----------------------------------------------------------------------------
```

Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation
Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.
Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.

3. **Description: Using the approve function of the token standard [Line 71]**

   The approve function of ERC-20 is vulnerable. Using a front-running attack one can spend approved tokens before the change of allowance value.

```
70
71 ▾    function approve(address spender, uint tokens) public returns (bool success) {
72          allowed[msg.sender][spender] = tokens;
73          emit Approval(msg.sender, spender, tokens);
74          return true;
75      }
```

   To prevent attack vectors described above, clients should make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. Though the contract itself shouldn't enforce it, to allow backward compatibility with contracts deployed before.

   Detailed reading around it can be found at EIP 20

4. **Description: Integer Overflow and Underflow [line 60]**

   If a balance reaches the maximum uint value (2^256) it will circle back to zero which checks for the condition. This may or may not be relevant, depending on the implementation. Think about whether or not the uint value has an opportunity to approach such a large number. Think about how the uint variable changes state, and who has authority to make such changes. If any user can call functions that update the uint value, it's more vulnerable to attack. If only an admin has access to change the variable's state, you might be safe. If a user can increment by only 1 at a time, you are probably also safe because there is no feasible way to reach this limit.
   The same is true for underflow. If a uint is made to be less than zero, it will cause an underflow and get set to its maximum value.
   Be careful with the smaller data-types like uint8, uint16, uint24...etc: they can even more easily hit their maximum value.

   Recommendations: One simple solution to mitigate the common mistakes for overflow and underflow is to use SafeMath.sol library for arithmetic functions. Also please read 20 cases for overflow and underflow.

5. **Description: Missing check on 'msg.data.length' could lead to short-address attack in this ERC20 transfer function. [line 77, 84]**

   This measure is to prevent The ERC20 Short Address Attack. this attack occurs when an attacker uses an incomplete address that ends with zeros 0,'00'...

   If an attacked generate an address like 0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa0000. he can provide 0xaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa (zeros omitted) to the transfer function This would cause a transfer of a value shifted by 16 bits, i.e. 65536 times larger than X, to the attacker's Ethereum account.

   Recommendations: Not to use that mitigation. One should not sign a transaction without validating the input parameters. Details about this issue can be read on this blog.

Automated Testing

We have used multiple automated testing frameworks. This makes code more secure common attacks. The results are below.

Slither

Slither is a Solidity static analysis framework that runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses. After running Slither we got the results below.

```
INFO:Detectors:
Pragma version^0.5.0 (BRUST.sol#1) allows old versions
solc-0.5.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solid
ity
INFO:Detectors:
Variable BURST._totalSupply (BRUST.sol#39) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformity-to-solidity-nami
ng-conventions
INFO:Detectors:
BURST.constructor() (BRUST.sol#49-57) uses literals with too many digits:
        - _totalSupply = 3100000000000000000000000 (BRUST.sol#53)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
totalSupply() should be declared external:
        - ERC20Interface.totalSupply() (BRUST.sol#8)
        - BURST.totalSupply() (BRUST.sol#59-61)
balanceOf(address) should be declared external:
        - ERC20Interface.balanceOf(address) (BRUST.sol#9)
        - BURST.balanceOf(address) (BRUST.sol#63-65)
allowance(address,address) should be declared external:
        - BURST.allowance(address,address) (BRUST.sol#67-69)
        - ERC20Interface.allowance(address,address) (BRUST.sol#10)
transfer(address,uint256) should be declared external:
        - ERC20Interface.transfer(address,uint256) (BRUST.sol#11)
        - BURST.transfer(address,uint256) (BRUST.sol#77-82)
approve(address,uint256) should be declared external:
        - ERC20Interface.approve(address,uint256) (BRUST.sol#12)
        - BURST.approve(address,uint256) (BRUST.sol#71-75)
transferFrom(address,address,uint256) should be declared external:
        - ERC20Interface.transferFrom(address,address,uint256) (BRUST.sol#13)
        - BURST.transferFrom(address,address,uint256) (BRUST.sol#84-90)
safeMul(uint256,uint256) should be declared external:
        - SafeMath.safeMul(uint256,uint256) (BRUST.sol#28)
safeDiv(uint256,uint256) should be declared external:
        - SafeMath.safeDiv(uint256,uint256) (BRUST.sol#28-30)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-de
clared-external
INFO:Slither:BRUST.sol analyzed (3 contracts with 46 detectors), 12 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Issues

1. solc-0.5.0 is not recommended for deployment.
   Reference read at this link.

2. Variable **BURST._totalSupply** (BRUST.sol#39) is not in **mixedCase**.
   Reference read at this link.

3. **BURST.constructor()** (BRUST.sol#49-57) uses literals with too many digits:
   - _totalSupply = 310000000000000000000000000 (BRUST.sol#53)

   Reference read at this link.

Manticore

Manticore is a symbolic execution tool for the analysis of smart contracts and binaries. It executes a program with symbolic inputs and explores all the possible states it can reach. It also detects crashes and other failure cases in binaries and smart contracts.
Manticore results throw the same warning which is similar to the Slither warning.

Mythx

MythX is a security analysis tool and API that performs static analysis, dynamic analysis, symbolic execution, and fuzzing on Ethereum smart contracts. MythX checks for and reports on the common security vulnerabilities in open industry-standard SWC Registry.

There are different functions within the whole file. I have separately put them for analysis. Most of the vulnerabilities generated by Mythx are discussed in Manual sections and slither. Mythx Report contains both Medium and low Severity issues.

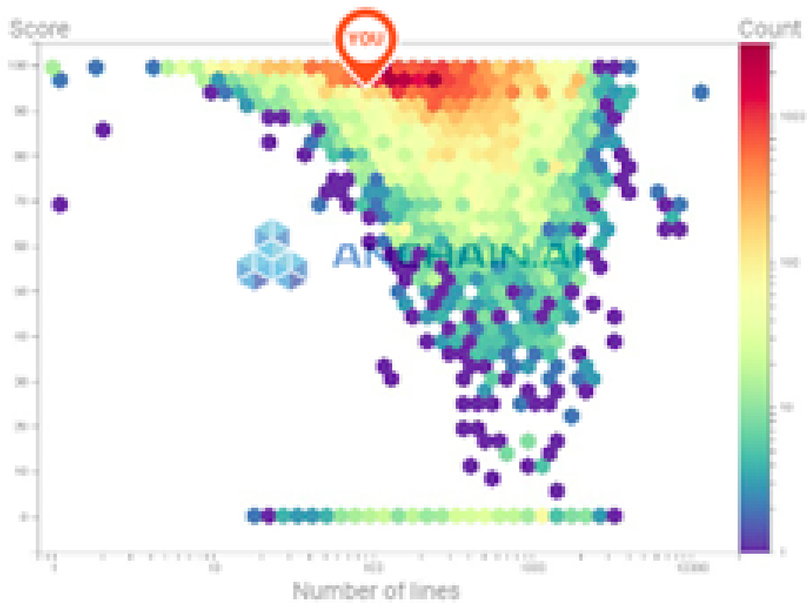
Anchain

Anchain sandbox audits the security score of any Solidity-based smart contract, having analyzed the source code of every mainnet EVM smart contract plus the 1M + unique, user-uploaded smart contracts. Code has been analyzed there and got the report below.

ETH Smart Contract Audit Report

MD5:1aa3a6064009757edf04909b74fc33ee Runtime:1.6s

Scored higher than
24% of similar code
amongst 50k smart contracts
audited by Anchain.AI .



Score
95.6

Threat Level
Low

Number of lines
91

Overview

Code Class	EVM Coverage
BURST	2.7%
ERC20Interface	0%

0 Vulnerabilities Found

High Risk Medium Risk Low Risk

Recommendations

- Consider using exact language version instead.
  See line(s) 1
- Missing check on 'msg.data.length' could lead to short-address attack in this ERC20 transfer function.
  See line(s) 77, 84
- Underflow or overflow may happen here, consider check boundaries such as assert(n < INT_MAX).
  See line(s) 60

Vulnerability Checklist

BURST

- Integer Underflow
- Integer Overflow
- Parity Multisig Bug
- Callstack Depth Attack
- Transaction-Ordering Dependency
- Timestamp Dependency
- Re-Entrancy

ERC20Interface

- Integer Underflow
- Integer Overflow
- Parity Multisig Bug
- Callstack Depth Attack
- Transaction-Ordering Dependency
- Timestamp Dependency
- Re-Entrancy

There were 0 vulnerabilities. However, there were some suggestions suggested by Anchain which are already discussed in the manual and automated audit.

Suggestions

1. Follow Open Zeppelin standard for token contract making. The contract written here is very simple token contract where some functions are missing like ERC20Burnable, ERC20Pausable, ERC20Mintable, TokenTimelock, IERC20 and a few other's. As a best practice, these also be there in the contract.

2. Missing License: Developers need to use a license according to your project. The suggestion to use here would be an SPDX license. You can find a list of licenses here: https://spdx.org/licenses/
   The SPDX License List is an integral part of the SPDX Specification. The SPDX License List itself is a list of commonly found licenses and exceptions used in free and open or collaborative software, data, hardware, or documentation. The SPDX License List includes a standardized short identifier, the full name, the license text, and a canonical permanent URL for each license and exception.

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and
a public bug bounty program to ensure the security of the code. Besides, a security audit, please don’t consider this report as investment advice.

Summary

The use of smart contracts is simple and the code is relatively small. Altogether the code is written and demonstrates effective use of abstraction, separation of concern, and modularity. But there are a few issues/vulnerabilities to be tackled at various security levels, it is recommended to fix them before deploying the contract on the main network. Given the subjective nature of some assessments, it will be up to the Burst team to decide whether any changes should be made.

Burst

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com