QuillAudits

Audit Report
February, 2021

Contents

Introduction

This Audit Report mainly focuses on the overall security of BSCrypt Token Lock Smart Contract. With this report, we have tried to ensure the reliability and correctness of their smart contract by complete and rigorous assessment of their system's architecture and the smart contract codebase.

Auditing Approach and Methodologies applied

The Quillhash team has performed rigorous testing of the project starting with analysing the code design patterns in which we reviewed the smart contract architecture to ensure it is well structured and safe use of third party smart contracts and libraries.

Our team then performed a formal line by line inspection of the Smart Contract to find any potential issue like race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

In the Unit testing Phase, we coded/conducted custom unit tests written for each function in the contract to verify that each function works as expected.

In Automated Testing, We tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was tested in collaboration of our multiple team members and this included -

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.
- Analysing the complexity of the code in depth and detailed, manual review of the code, line-by-line.
- Deploying the code on testnet using multiple clients to run live tests.
- Analysing failure preparations to check how the Smart Contract performs in case of any bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analysing the security of the on-chain data.

Audit Details

Project Name: BSCrypt Token Lock Smart Contract
Website/Etherscan
Code(Kovan):0xa47479D68cB75B9947Bbf373468171E045F86A1f
Languages: Solidity (Smart contract)
Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Securify, Mythril, Contract Library, Slither

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
- Quantity and quality of test coverage

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 level severity issues

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

Low level 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.

Informational

These are severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

	Gas Optimizations	High	Medium	Low	Recommendations
Open	6	0	2	0	10
Closed	0	0	0	0	1

Manual Audit

For this section the code was tested/read line by line by our developers. We also used Remix IDE's JavaScript VM and Kovan networks to test the contract functionality.

Gas Optimizations

1. The for loop within the **withdrawTokens** function to implement the pop functionality consumes a lot of unnecessary gas. **pop** has been added to built-in array support as of Pragma 0.5.0. If you intend to stick to the current pragma please refer to this stack overflow thread which provides multiple gas efficient alternatives which can suit your use case.

2. State variable .length of a non-memory array is used in the condition of for loop. In this case, every iteration of the loop consumes extra gas.

   Holding .length value in a local variable is more gas efficient.

3. Contract **owned** need not be inherited within **lockToken** contract as it is not being used within it.

4. **allDepositIds** is not very useful and is recommended to be removed as all the deposit IDs are in a range and we will just be storing range numbers which can also be easily calculated using variable **depositId**.

5. Either we should remove all the getter functions within **lockToken** contracts as all the state variables exposed by them are defined with public visibility and will already be exposed. OR we can change the visibility of all the state variables exposed by getters to private.

   Note: We can also change to internal visibility if in future the contract would be inherited and the state variables will be used in the child contract.

6. A function with a **public** visibility modifier that is not called internally should be set to external visibility to increase code readability. Moreover, in many cases, functions with external visibility modifier spend less gas compared to functions with public visibility modifier. Following functions can be declared external:

    - transferOwnership
    - getDepositDetails
    - getDepositsByWithdrawalAddress
    - getTotalTokenBalance
    - lockTokens
    - getAllDepositIds
    - getTokenBalanceByAddress
    - withdrawTokens

Low level severity issues

1. None

Medium severity issues

1. Unsafe array's length manipulation:
   depositsByWithdrawalAddress[lockedToken[_id].withdrawalAddress].length--
   The length of the dynamic array is changed directly. In this case, the appearance of gigantic arrays is possible and it can lead to a storage overlap attack (collisions with other data in storage).

2. In the **withdrawTokens** function, following line:

   require(Token(lockedToken[_id].tokenAddress).transfer(msg.sender, lockedToken[_id].tokenAmount), 'Transfer of tokens failed');

   Makes an external call via **transfer** function. Since the **tokenAddress** can also be custom and can be set via **lockTokens** function, this external call is untrusted. And can be used to introduce re-entrancy. We recommend moving the transfer external call after following two lines to avoid this:

   require(!lockedToken[_id].withdrawn, 'Tokens already withdrawn');
   lockedToken[_id].withdrawn = true;

Recommendations

1. The Token contract should be turned into an interface as it does not implement any function declared within it.

2. We recommend adding proper error messages to require statements to improve readability and debugging of contracts.

3. Names of contract **lockToken**, owned should be updated to LockToken and Owned as per Solidity's style guide recommendation.

4. The visibility modifier **onlyOwner** for function **transferOwnership** should come before any custom modifiers as per Solidity style guide.

5. Commenting can be added for many functions and existing commenting can be updated as per Solidity's recommendation by using NatSpec.

6. **lockToken** contract does not use custom events specific to the contract functionality. Events should be fired with all state variable updates as good practice. This makes it easier to build dApps on top of the contract's using existing tools. For instance when the owner is changed or when the contract receives ethers.

7. It is recommended to add the following line at the beginning of contract code, in order to add basic licensing to the contract:
   // SPDX-License-Identifier: MIT

8. Compiler version should be fixed:
   Solidity source files indicate the versions of the compiler they can be compiled with.
   pragma solidity ^0.4.17; // bad: compiles w 0.4.17 and above
   pragma solidity 0.4.24; // good : compiles w 0.4.24 only

It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

9. Pragma version(^0.4.25) allows old versions, solc-0.4.26 is not recommended for the deployment of the smart contract. Solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

    Deploy with any of the following Solidity versions:

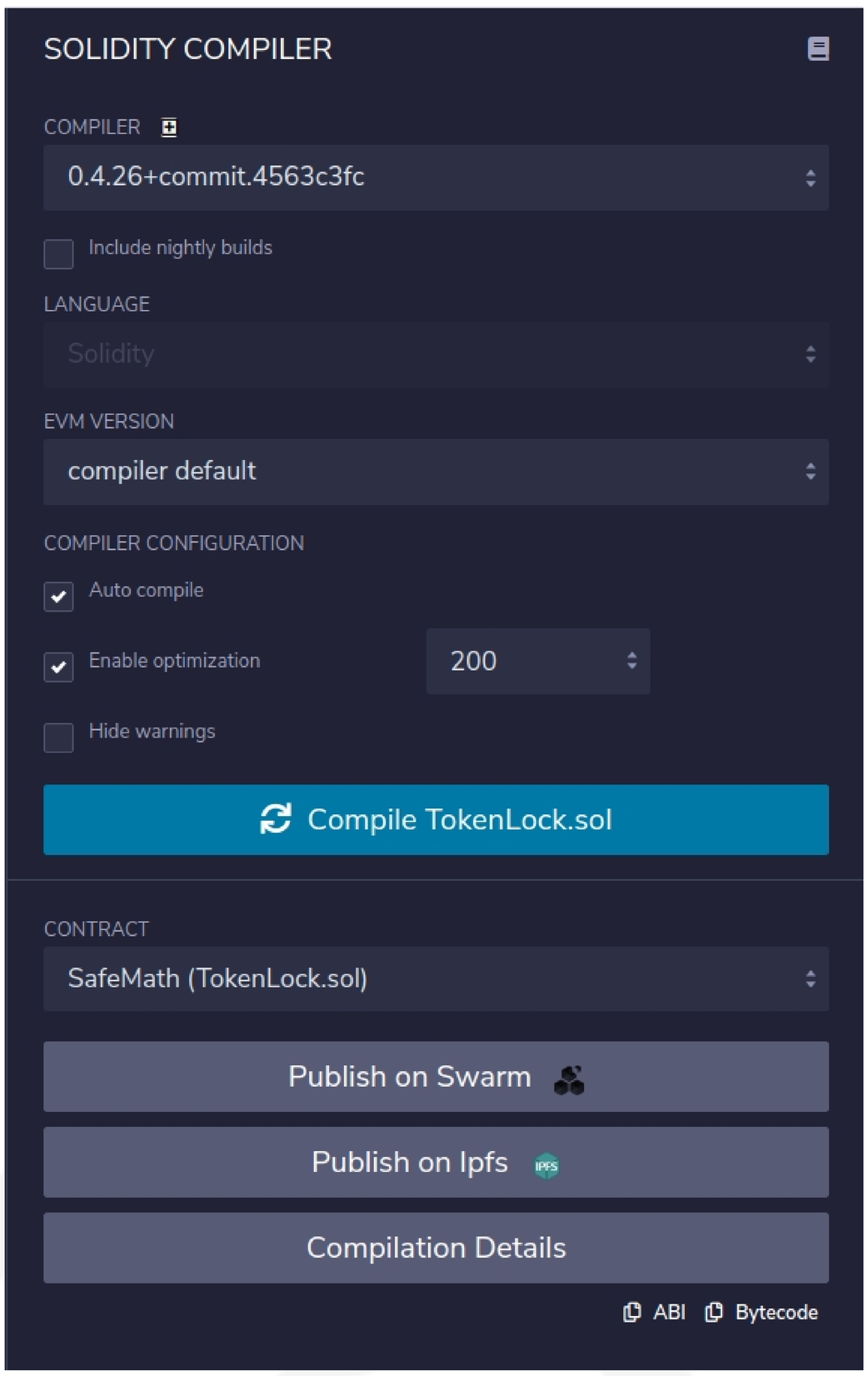
    - 0.5.11 - 0.5.13,
    - 0.5.15 - 0.5.17,
    - 0.6.8,
    - 0.6.10 - 0.6.11. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

10. The Token contract is not standard ERC20, has an extra **approveAndCall** function which has not been used again in the **lockToken** contract. For a standard interface implementation of ERC20 refer OpenZeppelin IERC20.sol.

Automated Testing

Remix Compiler Warnings

It throws warnings by Solidity's compiler. If it encounters any errors the contract cannot be compiled and deployed.



No warning Or errors were emitted by Remix.

Solhint Linting Violations

Solhint is an open-source project for linting solidity code, providing both security and style guide validations. It integrates seamlessly into most mainstream IDEs. We used Solhint as a plugin within our VScode for this analysis. Multiple Linting violations were detected by Solhint, the linted version of the contract is as follows:

```
pragma solidity ^0.4.25;

contract Token {
  function balanceOf(address who) external view returns (uint256);

  function allowance(address owner, address spender)
    external
    view
    returns (uint256);

  function transfer(address to, uint256 value) external returns (bool);

  function approve(address spender, uint256 value) external returns (bool);

  function approveAndCall(
    address spender,
    uint256 tokens,
    bytes data
  ) external returns (bool success);

  function transferFrom(
    address from,
    address to,
    uint256 value
  ) external returns (bool);
}

library SafeMath {
  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    require(c / a == b);
    return c;
}

function div(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a / b;
    return c;
}

function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);
    return a - b;
}

function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a);
    return c;
  }

  function ceil(uint256 a, uint256 m) internal pure returns (uint256) {
    uint256 c = add(a, m);
    uint256 d = sub(c, 1);
    return mul(div(d, m), m);
  }
}
```

```
contract owned {
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        owner = newOwner;
    }
}

contract lockToken is owned {
    using SafeMath for uint256;

    /*
     * deposit vars
     */
    struct Items {
        address tokenAddress;
        address withdrawalAddress;
        uint256 tokenAmount;
        uint256 unlockTime;
        bool withdrawn;
    }

    uint256 public depositId;
    uint256[] public allDepositIds;
    mapping(address => uint256[]) public depositsByWithdrawalAddress;
    mapping(uint256 => Items) public lockedToken;
    mapping(address => mapping(address => uint256)) public walletTokenBalance;

    event LogWithdrawal(address SentToAddress, uint256 AmountTransferred);

    /**
     * Constrctor function
     */
    constructor() public {}

    /**
     *lock tokens
     */
    function lockTokens(
        address _tokenAddress,
        uint256 _amount,
        uint256 _unlockTime
    ) public returns (uint256 _id) {
        require(_amount > 0, "token amount is Zero");
        require(
            _unlockTime < 10000000000,
            "Enter an unix timestamp in seconds, not miliseconds"
        );
        require(
            Token(_tokenAddress).approve(this, _amount),
            "Approve tokens failed"
        );
        require(
            Token(_tokenAddress).transferFrom(msg.sender, this, _amount),
```

```
"Transfer of tokens failed"
        );

        //update balance in address
        walletTokenBalance[_tokenAddress][msg.sender] = walletTokenBalance[
            _tokenAddress
        ][msg.sender]
            .add(_amount);

        address _withdrawalAddress = msg.sender;
        _id = ++depositId;
        lockedToken[_id].tokenAddress = _tokenAddress;
        lockedToken[_id].withdrawalAddress = _withdrawalAddress;
        lockedToken[_id].tokenAmount = _amount;
        lockedToken[_id].unlockTime = _unlockTime;
        lockedToken[_id].withdrawn = false;

        allDepositIds.push(_id);
        depositsByWithdrawalAddress[_withdrawalAddress].push(_id);
    }

    /**
     *withdraw tokens
     */
    function withdrawTokens(uint256 _id) public {
        require(
            block.timestamp >= lockedToken[_id].unlockTime,
            "Tokens are locked"
        );
        require(
            msg.sender == lockedToken[_id].withdrawalAddress,
            "Can withdraw by withdrawal Address only"
        );
        require(!lockedToken[_id].withdrawn, "Tokens already withdrawn");
        require(
            Token(lockedToken[_id].tokenAddress).transfer(
                msg.sender,
                lockedToken[_id].tokenAmount
            ),
            "Transfer of tokens failed"
        );

        lockedToken[_id].withdrawn = true;

        //update balance in address
        walletTokenBalance[lockedToken[_id].tokenAddress][
            msg.sender
        ] = walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(
            lockedToken[_id].tokenAmount
        );

        //remove this id from this address
        uint256 i;
        uint256 j;
        for (
            j = 0;
            j <
            depositsByWithdrawalAddress[lockedToken[_id].withdrawalAddress]
                .length;
            j++
    ) {
            if (
```

```
            depositsByWithdrawalAddress[lockedToken[_id].withdrawalAddress][
                j
            ] == _id
        ) {
            for (
                i = j;
                i <
                depositsByWithdrawalAddress[
                    lockedToken[_id].withdrawalAddress
                ]
                    .length -
                    1;
                i++
            ) {
                depositsByWithdrawalAddress[
                    lockedToken[_id].withdrawalAddress
                ][i] = depositsByWithdrawalAddress[
                    lockedToken[_id].withdrawalAddress
                ][i + 1];
            }
            depositsByWithdrawalAddress[lockedToken[_id].withdrawalAddress]
                .length--;
            break;
        }
    }
    emit LogWithdrawal(msg.sender, lockedToken[_id].tokenAmount);
}

/*get total token balance in contract*/
function getTotalTokenBalance(address _tokenAddress)
    public
    view
    returns (uint256)
{
    return Token(_tokenAddress).balanceOf(this);
}

/*get total token balance by address*/
function getTokenBalanceByAddress(
    address _tokenAddress,
    address _walletAddress
) public view returns (uint256) {
    return walletTokenBalance[_tokenAddress][_walletAddress];
}

/*get allDepositIds*/
function getAllDepositIds() public view returns (uint256[]) {
    return allDepositIds;
}

/*get getDepositDetails*/
function getDepositDetails(uint256 _id)
    public
    view
    returns (
        address,
        address,
        uint256,
        uint256,
        bool
    )
{
```

```
    return (
        lockedToken[_id].tokenAddress,
        lockedToken[_id].withdrawalAddress,
        lockedToken[_id].tokenAmount,
        lockedToken[_id].unlockTime,
        lockedToken[_id].withdrawn
    );
}

/*get DepositsByWithdrawalAddress*/
function getDepositsByWithdrawalAddress(address _withdrawalAddress)
    public
    view
    returns (uint256[])
  {
    return depositsByWithdrawalAddress[_withdrawalAddress];
  }
}
```

To understand the changes recommended here, please refer Solidity's Style Guide.

Contract Library

Contract-library contains the most complete, high-level decompiled representation of all Ethereum smart contracts, with security analysis applied to these in real time.

We performed analysis using contract Library on the mainnet address of the BSCrypt contract: 0xa47479D68cB75B9947Bbf373468171E045F86A1f

Analysis summary can be accessed here: https://contractlibrary.com/contracts/Kovan/0xa47479d68cb75b9947bbf373468171e045f86a1f

The concerns it raises have already been covered in the manual audit section.

Slither

Slither, an open-source static analysis framework. This tool provides rich information about Ethereum smart contracts and has the critical properties. While Slither is built as a security-oriented static analysis framework, it is also used to enhance the user's understanding of smart contracts, assist in code reviews, and detect missing optimizations.

The concerns it raises have already been covered in the manual audit section.

```
INFO:Detectors:
Reentrancy in lockToken.withdrawTokens(uint256) (tokenLock.sol#118-141):
	External calls:
	- require(bool,string)
(Token(lockedToken[_id].tokenAddress).transfer(msg.sender,lockedToken[_id].tokenAmount),Transfer of
tokens failed) (tokenLock.sol#122)
	State variables written after the call(s):
	- lockedToken[_id].withdrawn = true (tokenLock.sol#124)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
Reentrancy in lockToken.lockTokens(address,uint256,uint256) (tokenLock.sol#94-113):
	External calls:
	- require(bool,string)(Token(_tokenAddress).approve(this,_amount),Approve tokens failed)
(tokenLock.sol#97)
	- require(bool,string)(Token(_tokenAddress).transferFrom(msg.sender,this,_amount),Transfer of tokens
failed) (tokenLock.sol#98)
	State variables written after the call(s):
	- allDepositIds.push(_id) (tokenLock.sol#111)
	- _id = ++ depositId (tokenLock.sol#104)
	- depositsByWithdrawalAddress[_withdrawalAddress].push(_id) (tokenLock.sol#112)
	- lockedToken[_id].tokenAddress = _tokenAddress (tokenLock.sol#105)
	- lockedToken[_id].withdrawalAddress = _withdrawalAddress (tokenLock.sol#106)
	- lockedToken[_id].tokenAmount = _amount (tokenLock.sol#107)
	- lockedToken[_id].unlockTime = _unlockTime (tokenLock.sol#108)
	- lockedToken[_id].withdrawn = false (tokenLock.sol#109)
	- walletTokenBalance[_tokenAddress][msg.sender] = walletTokenBalance[_tokenAddress]
[msg.sender].add(_amount) (tokenLock.sol#101)
Reentrancy in lockToken.withdrawTokens(uint256) (tokenLock.sol#118-141):
	External calls:
	- require(bool,string)
(Token(lockedToken[_id].tokenAddress).transfer(msg.sender,lockedToken[_id].tokenAmount),Transfer of
tokens failed) (tokenLock.sol#122)
	State variables written after the call(s):
	- depositsByWithdrawalAddress[lockedToken[_id].withdrawalAddress][i] =
depositsByWithdrawalAddress[lockedToken[_id].withdrawalAddress][i + 1] (tokenLock.sol#134)
	- depositsByWithdrawalAddress[lockedToken[_id].withdrawalAddress].length -- (tokenLock.sol#136)
	- walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender] =
walletTokenBalance[lockedToken[_id].tokenAddress][msg.sender].sub(lockedToken[_id].tokenAmount)
(tokenLock.sol#127)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in lockToken.withdrawTokens(uint256) (tokenLock.sol#118-141):

External calls:
	- require(bool,string)
(Token(lockedToken[_id].tokenAddress).transfer(msg.sender,lockedToken[_id].tokenAmount),Transfer of
tokens failed) (tokenLock.sol#122)
	Event emitted after the call(s):
	- LogWithdrawal(msg.sender,lockedToken[_id].tokenAmount) (tokenLock.sol#140)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
lockToken.withdrawTokens(uint256) (tokenLock.sol#118-141) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(block.timestamp >= lockedToken[_id].unlockTime,Tokens are locked)
(tokenLock.sol#119)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Pragma version^0.4.25 (tokenLock.sol#1) allows old versions
solc-0.4.26 is not recommended for deployment
```

```
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Contract owned (tokenLock.sol#45-60) is not in CapWords
Contract lockToken (tokenLock.sol#62-174) is not in CapWords
Parameter lockToken.lockTokens(address,uint256,uint256)._tokenAddress (tokenLock.sol#94) is not in
mixedCase
Parameter lockToken.lockTokens(address,uint256,uint256)._amount (tokenLock.sol#94) is not in mixedCase
Parameter lockToken.lockTokens(address,uint256,uint256)._unlockTime (tokenLock.sol#94) is not in mixedCase
Parameter lockToken.withdrawTokens(uint256)._id (tokenLock.sol#118) is not in mixedCase
Parameter lockToken.getTotalTokenBalance(address)._tokenAddress (tokenLock.sol#144) is not in mixedCase
Parameter lockToken.getTokenBalanceByAddress(address,address)._tokenAddress (tokenLock.sol#150) is not
in mixedCase
Parameter lockToken.getTokenBalanceByAddress(address,address)._walletAddress (tokenLock.sol#150) is not
in mixedCase
Parameter lockToken.getDepositDetails(uint256)._id (tokenLock.sol#162) is not in mixedCase
Parameter lockToken.getDepositsByWithdrawalAddress(address)._withdrawalAddress (tokenLock.sol#169) is
not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformity-to-solidity-naming-
conventions
INFO:Detectors:
lockToken.lockTokens(address,uint256,uint256) (tokenLock.sol#94-113) uses literals with too many digits:
        - require(bool,string)(_unlockTime < 10000000000,Enter an unix timestamp in seconds, not
miliseconds) (tokenLock.sol#96)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
transferOwnership(address) should be declared external:
        - owned.transferOwnership(address) (tokenLock.sol#57-59)
lockTokens(address,uint256,uint256) should be declared external:
        - lockToken.lockTokens(address,uint256,uint256) (tokenLock.sol#94-113)
withdrawTokens(uint256) should be declared external:
        - lockToken.withdrawTokens(uint256) (tokenLock.sol#118-141)
getTotalTokenBalance(address) should be declared external:
        - lockToken.getTotalTokenBalance(address) (tokenLock.sol#144-147)
getTokenBalanceByAddress(address,address) should be declared external:
        - lockToken.getTokenBalanceByAddress(address,address) (tokenLock.sol#150-153)
getAllDepositIds() should be declared external:
        - lockToken.getAllDepositIds() (tokenLock.sol#156-159)
getDepositDetails(uint256) should be declared external:
        - lockToken.getDepositDetails(uint256) (tokenLock.sol#162-166)
getDepositsByWithdrawalAddress(address) should be declared external:
        - lockToken.getDepositsByWithdrawalAddress(address) (tokenLock.sol#169-172)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-
declared-external
INFO:Slither:. analyzed (4 contracts with 46 detectors), 27 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the BSCrypt Token Lock Smart Contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Summary

Use case of the smart contract is simple and the code is relatively small. Altogether, the code is written and demonstrates effective use of abstraction, separation of concerns, and modularity. There are a number of issues/vulnerabilities to be tackled in various severity levels, but none of them is of high severity.

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

hello@quillhash.com