BQ Labs

SMART CONTRACT AUDIT REPORT

Prepared by:

BlockAudit

Date Of Enrollment:

November 21, 2024 - December 02, 2024

www.blockaudit.report

TABLE OF CONTENTS

SUMMARY

This Audit Report mainly focuses on the extensive security of **BQ Labs** Smart Contracts. With this report, we attempt to ensure the reliability and correctness of the smart contract by complete and rigorous assessment of the system's architecture and the smart contract codebase.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts producedby industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified inpublic;
- Provide more transparency on privileged activities once the protocol is live.

OVERVIEW

Project Summary

Project Name	BQ Labs
Language	Solidity
Platform	---
Commit Hash	b17b0934b92b840942a505915078aec33ba33fd4

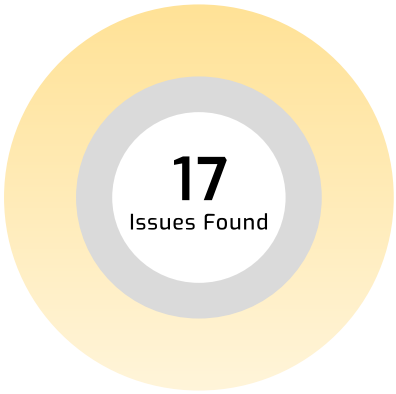
File Summary

ID	File Name
BTC	bqBTC.sol
COVERLIB	CoverLib.sol
INSURANCE	InsurancePool.so
TOKEN	BQToken.sol
GOV	Gov.sol
COVER	InsuranceCover.sol

Date of Delivery	2 Dec 2024
Audit Methodology	Code Analysis. Automatic Assesment, Manual Review
Audit Result	**Passed** ✓✓
Audit Team	BlockAudit Report Team

FINDINGS

Critical	2	12%
High	0	0.0%
Medium	3	18%
Low	3	18%
Gas	6	34%
Informational	3	18%



Vulnerability Findings Summary (Part 1)

ID	Type	Severity	Status
C-01	Lack Of Amount Validation In Mint Function	Critical	**Fixed**
C-02	No Implementation of conversion rate for minting bqBTC based on Native Token	Critical	**Fixed**
M-01	User pool deposits not deducted in withdraw function of Pool Contract	Medium	**Fixed**
M-02	Centralization Risk for trusted owners	Medium	**Fixed**
M-03	Use SafeERC20 Library	Medium	**Fixed**
L-01	Avoid using Floating Pragma	Low	**Fixed**
L-02	Use a more recent version of solidity	Low	**Fixed**
L-03	Lack of 2-step transfer of ownership	Low	**Fixed**

Vulnerability Findings Summary (Part 2)

ID	Type	Severity	Status
G-01	Cache Array Length Outside Of Loop	Gas Error	**Acknowledged**
G-02	Use Custom Errors instead of Revert Strings to save Gas	Gas Error	**Acknowledged**
G-03	Use ++i Instead of i++ or i += 1 for Gas Efficiency	Gas Error	**Acknowledged**
G-04	Splitting require() statements that use && saves gas	Gas Error	**Acknowledged**
G-05	Increments/decrements can be unchecked in for-loops	Gas Error	**Acknowledged**
G-06	Use != 0 instead of > 0 for unsigned integer comparison	Gas Error	**Acknowledged**
I-01	Use a modifier instead of a require/if statement for a special msg.sender actor	Informational	**Acknowledged**
I-02	Variables need not be initialized to zero	Informational	**Acknowledged**
I-03	Missing Natspec	Informational	**Acknowledged**

C-01

Type	Lack of Mint Amount Validation in mint Function
Severity	Critical
File	bqBTC.sol
Instances	37
Status	**Acknowledged**

Description

The mint function in the bqBTC contract allows users to mint a specified amount of bqBTC tokens based on either native cryptocurrency (BNB) or BTC tokens. The is no check on if the amount parameter, which specifies how many tokens to mint, is valid or reasonable. This can lead to unintended consequences, such as excessive token minting.

Snapshot

```
ftrace | funcSig
37    function mint(
38        address account⬆,
39        uint256 amount⬆,
40        uint256 btcAmount⬆
41    ) external payable {
42        bool nativeSent = msg.value >= minMintAmount;
43        bool btcSent = false;
44
45        if (!nativeSent) {
46            require(
47                btcAmount⬆ >= minMintAmount,
48                "amount must be greater than 0"
49            );
50            require(
51                bscBTC.transferFrom(msg.sender, address(this), btcAmount⬆),
52                "Insufficient BTC tokens sent to mint"
53            );
54            btcSent = true;
55        }
56        require(nativeSent || btcSent, "Insufficient tokens sent to mint");
57        _mint(account⬆, amount⬆);
58    }
59
```

Impact:

The function checks if the msg.value (native currency, eg. BNB token) >= minMintAmount. If not, it requires that btcAmount >= minMintAmount and that the transfer of BTC tokens is successful. The function then mints tokens to the specified account without validating the amount parameter.
An attacker can send a large value (1000e18) for the amount parameter while satisfying either the native currency or BTC token requirement.

Proof of Concept:

To demonstrate this vulnerability, consider the following scenario:

1. A user calls the mint function with:
   - minMintAmount = 1e18 (1 BTC)
   - btcAmount = 2e18 (2 BTC)
   - amount = 100e18 (100 bqBTC)
2. Since btcAmount exceeds minMintAmount, the condition for minMintAmount is satisfied (nativeSent = true).
3. The function proceeds to call _mint(account, amount) without validating whether amount is within acceptable limits.

Recommended Mitigation:

Add a proper input sanitization.

C-02

Type	No Implementation of conversion rate for minting bqBTC based on Native Token
Severity	Critical
File	bqBTC.sol
Instances	37
Status	**Acknowledged**

Description

In addition to the lack of validation for the amount parameter in the mint function, there is a significant concern regarding Native Minting functionality. For Example: In the BNB Chain, the current implementation does not have any conversion rate of minting bqBTC based on Native token (For example, BNB) they send. This can lead to unintended consequences and potential exploitation.

Snapshot

```
37:     function mint(
38:         address account,
39          uint256 amount,
40          uint256 btcAmount
41      ) external payable {
42          bool nativeSent = msg.value >= minMintAmount;
43          bool btcSent = false;
44
45          if (!nativeSent) {
46              require(
47                  btcAmount >= minMintAmount,
48                  "amount must be greater than 0"
49              );
50              require(
51                  bscBTC.transferFrom(msg.sender, address(this),
btcAmount),
52                  "Insufficient BTC tokens sent to mint"
53              );
54              btcSent = true;
55          }
56          require(nativeSent || btcSent, "Insufficient tokens sen
t to mint");
57          _mint(account, amount);
58      }
```

Impact

When users send BNB to the mint function, the contract mints an amount of bqBTC tokens without any checks on the relationship between BNB value and the minted amount. This creates several risks like Token Devaluation, Minting large supply of bqBTC, etc.

POC:

Consider the following scenario:

1. A user sends 1 BNB (equivalent to 1e18) as msg.value. (Suppose minMintAmount is 1 BNB)
2. The function mints allows user to mint any number bqBTC tokens for the user without any additional checks or limits.

This lack of conversion from Native token (eg. BNB) to bqBTC means that if BNB's value increases significantly or if there is a sudden influx of users sending large amounts of BNB, it could result in excessive minting of bqBTC tokens.

Recommended Mitigation Steps:

Implement a mechanism that adjusts the conversion rate based on real-time market data or predefined limits.

M-01

Type	User pool deposits not deducted in withdraw function of Pool Contract
Severity	Medium
File	InsurancePool.sol
Instances	342
Status	**Acknowledged**

Description

When the user calls the Deposit function in the Insurance Pool contract, the function updates the selectedPool.deposits[msg.sender].amount to the deposited amount. But, The withdraw function does not update the userDeposit.amount after a withdrawal occurs. This oversight can lead to inconsistencies in the state of the pool, particularly regarding user deposits, which may cause confusion and potential glitch in the frontend interface. It is considered best practice to ensure that all state changes in a smart contract are explicitly updated and reflected accurately.

Snapshot:

```
  342:     function deposit(uint256 _poolId, uint256 _amount) public nonReentran
t {
  343         Pool storage selectedPool = pools[_poolId];
  344
  345         require(_amount > 0, "Amount must be greater than 0");
  346         require(selectedPool.isActive, "Pool is inactive or does not exis
t");
  347
  348         bqBTC.burn(msg.sender, _amount);
  349         selectedPool.tvl += _amount;
  350
  351         if (selectedPool.deposits[msg.sender].amount > 0) {
  352             uint256 amount = selectedPool.deposits[msg.sender].amount + _
amount;
  353             selectedPool.deposits[msg.sender].amount = amount;
  354             selectedPool.deposits[msg.sender].expiryDate =
  355                 block.timestamp +
  356                 (selectedPool.minPeriod * 1 days);
  357             selectedPool.deposits[msg.sender].startDate = block.timestam
p;
  358             selectedPool.deposits[msg.sender].dailyPayout =
  359                 (amount * selectedPool.apy) /
  360                 100 /
  361                 365;
  362             selectedPool.deposits[msg.sender].daysLeft = (selectedPool
  363                 .minPeriod * 1 days);
  364         } else {
  365             uint256 dailyPayout = (_amount * selectedPool.apy) / 100 / 36
5;
  366             selectedPool.deposits[msg.sender] = Deposits({
  367                 lp: msg.sender,
  368                 amount: _amount,
  369                 poolId: _poolId,
  370                 dailyPayout: dailyPayout,
  371                 status: Status.Active,
  372                 daysLeft: selectedPool.minPeriod,
  373                 startDate: block.timestamp,
  374                 expiryDate: block.timestamp +
  375                     (selectedPool.minPeriod * 1 minutes),
  376                 accruedPayout: 0
  377             });
  378         }
```

Snapshot:

JavaScript Copy

```
    function withdraw(uint256 _poolId) public nonReentrant {
        Pool storage selectedPool = pools[_poolId];
        Deposits storage userDeposit = selectedPool.deposits[msg.sender];

        require(userDeposit.amount > 0, "No deposit found for this addres
s");
        require(userDeposit.status == Status.Active, "Deposit is not acti
ve");
        require(
            block.timestamp >= userDeposit.expiryDate,
            "Deposit period has not ended"
        );

        userDeposit.status = Status.Withdrawn;
        selectedPool.tvl -= userDeposit.amount;
        CoverLib.Cover[] memory poolCovers = getPoolCovers(_poolId);
        for (uint i = 0; i < poolCovers.length; i++) {
            ICoverContract.updateMaxAmount(poolCovers[i].id);
        }

        bqBTC.bqMint(msg.sender, userDeposit.amount);

        emit Withdraw(msg.sender, userDeposit.amount, selectedPool.poolNa
me);
    }
```

Recommended Mitigation Steps:

To rectify this issue, update the withdraw function to deduct or reset userDeposit.amount after a successful withdrawal:

```
userDeposit.amount = 0; // Resetting amount after withdrawal
```

M-02

Type	Centralization Risk for trusted owners
Severity	Medium
File	-----
Instances	5, 6
Status	**Acknowledged**

Description

In the contracts, There are multiple function where admin/ owner can perform critical actions that can possess as a significant centralization risk. like setVotingDuration, updateRewardAmount, deactivatePool, adding or removing admin etc.
The contracts exhibit multiple functions that grant the admin or owner the ability to execute critical actions, which poses a significant centralization risk. Key functions include setVotingDuration, updateRewardAmount, deactivatePool, and the ability to add or remove administrators.

Impact:

The concentration of control in a limited number of individuals heightens the risk of malicious actions or inadvertent errors, potentially jeopardizing user funds and undermining trust in the protocol.

Recommended Mitigation Steps:

Transition from using Externally Owned Accounts (EOAs) for governance functions to a multi-signature wallet. This approach requires multiple signatures to execute critical actions, significantly reducing the risk of unilateral decision-making and enhancing security.
Avoid hardcoding addresses in the contract. Instead, consider using a more flexible approach such as allowing an admin to add or remove other admins through a controlled process.

M-03

Type	Use SafeERC20 Library
Severity	Medium
File	bqBTC.sol
Instances	51
Status	**Acknowledged**

Description

ERC20 standard allows the transfer function of some contracts to return bool or return nothing. Some tokens such as USDT return nothing.
This could lead to funds stuck in the contract without the possibility to retrieve them. Using safeTransferFrom of SafeERC20.sol is recommended instead.
For example Tether (USDT)'s transfer() and transferFrom() functions on L1 do not return booleans as the specification requires, and instead have no return value. When these sorts of tokens are cast to IERC20, their function signatures do not match and therefore the calls made, revert (see this link for a test case). The number of affected tokens is higher than one would expect, due to the fact that at one point OpenZeppelin was using the incorrect signatures in its published interface. Use
OpenZeppelin's SafeERC20's safeTransfer()/safeTransferFrom() instead

Snapshot:

```
51:         bscBTC.transferFrom(msg.sender, address(this), btcAmount),
84:         return super.transfer(to, amount);
92:         return super.transferFrom(from, to, amount);
```

Recommended Mitigation Steps:

We recommend using OpenZeppelin's SafeERC20 versions with the safeTransfer and safeTransferFrom functions that handle the return value check as well as non-standard-compliant tokens.

L-01

Type	Avoid Using Floating Pragma
Severity	Low
File	All
Instances	------
Status	**Acknowledged**

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with 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.

Snapshot

```
Solidity
bqBTC.sol
  2: pragma solidity ^0.8.0;

BQToken.sol
  2: pragma solidity ^0.8.0;

CoverLib.sol
  2: pragma solidity ^0.8.0;

Gov.sol
  2: pragma solidity ^0.8.0;

InsuranceCover.sol
  2: pragma solidity ^0.8.0;

InsurancePool.sol
  2: pragma solidity ^0.8.0;
```

Recommended Mitigation Steps:

Consider replacing ^0.8.0 by 0.8.26

L-02

Type	Use a more recent version of solidity
Severity	Low
File	All
Instances	-------
Status	**Acknowledged**

Description

When deploying contracts, you should use the latest released version of Solidity. Apart from exceptional cases, only the latest version receives security fixes. Furthermore, breaking changes, as well as new features, are introduced regularly.

Snapshot:

```
/Users/rohan16/Desktop/smartcontract/contracts/bqBTC.sol
2: pragma solidity ^0.8.0;

2: pragma solidity ^0.8.0;

/Users/rohan16/Desktop/smartcontract/contracts/CoverLib.sol
2: pragma solidity ^0.8.0;

/Users/rohan16/Desktop/smartcontract/contracts/Gov.sol
2: pragma solidity ^0.8.0;

/Users/rohan16/Desktop/smartcontract/contracts/InsuranceCover.sol
2: pragma solidity ^0.8.0;

/Users/rohan16/Desktop/smartcontract/contracts/InsurancePool.sol
2: pragma solidity ^0.8.0;
```

Recommended Mitigation:

Consider using the latest stable release i.e. 0.8.26

L-03

Type	Lack of 2-step transfer of ownership
Severity	Low
File	bqBTC.sol, Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	-------
Status	**Acknowledged**

Description

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met). Custom errors are defined using the error statement, which can be used inside and outside of contracts (including interfaces and libraries).

Snapshot:

```
contracts/bqBTC.sol:
  8: contract bqBTC is ERC20, Ownable {

contracts/Gov.sol:
  47: contract Governance is ReentrancyGuard, Ownable {

contracts/InsuranceCover.sol:
  91: contract InsuranceCover is ReentrancyGuard, Ownable {

contracts/InsurancePool.sol:
  68: contract InsurancePool is ReentrancyGuard, Ownable {
```

Recommended Mitigation:

Recommend considering implementing a two-step process where the owner nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of ownership to fully succeed.

G-01

Type	Cache array length outside of loop
Severity	Gas Error
File	Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	--------
Status	**Acknowledged**

Description

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

```
Gov.sol
  151:         for (uint256 i = 1; i < proposalCounter; i++) {
  180:         for (uint i = 0; i < participants.length; i++) {
  239:         for (uint i = 0; i < participants.length; i++) {
  255:         for (uint256 i = 0; i < proposalIds.length; i++) {
  330:         for (uint256 i = 0; i < proposalIds.length; i++) {
  345:         for (uint256 i = 0; i < proposalIds.length; i++) {
  356:         for (uint256 i = 0; i < proposalIds.length; i++) {
  381:         for (uint256 i = 0; i < proposalIds.length; i++) {
  391:         for (uint256 i = 0; i < proposalIds.length; i++) {

InsuranceCover.sol
  208:         for (uint256 i = 0; i < coversInPool.length; i++) {
  269:         for (uint256 i = 0; i < coversInPool.length; i++) {
  357:         for (uint i = 0; i < participants.length; i++) {
  378:         for (uint256 i = 0; i < coverIds.length; i++) {
  389:         for (uint256 i = 0; i < coverIds.length; i++) {
  406:         for (uint256 i = 0; i < coverIds.length; i++) {
  418:         for (uint256 i = 0; i < coverIds.length; i++) {
  452:         for (uint256 i = 1; i < coverIds.length; i++) {

InsurancePool.sol
  236:         for (uint256 i = 1; i <= poolCount; i++) {
  257:         for (uint i = 0; i < poolToCovers[_poolId].length; i++) {
  282:         for (uint256 i = 1; i <= poolCount; i++) {
  293:         for (uint256 i = 1; i <= poolCount; i++) {
  333:         for (uint i = 0; i < poolCovers.length; i++) {
  381:         for (uint i = 0; i < poolCovers.length; i++) {
  386:         for (uint i = 0; i < participants.length; i++) {
  424:         for (uint i = 0; i < poolCovers.length; i++) {
```

Recommended Mitigation Steps:

```
// Original code
for (uint i = 0; i < poolCovers.length; i++) {

// Refactored code
uint256 coversLength = poolCovers.length;
for (uint i = 0; i < coversLength; i++) {
    // Loop logic here
}
```

G-02

Type	Use Custom Errors instead of Revert Strings to save Gas
Severity	Gas Error
File	Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	------
Status	**Acknowledged**

Description

Starting from Solidity version 0.8.4, developers can utilize custom errors as a more gas-efficient alternative to revert strings. Custom errors save approximately 50 gas each time they are triggered by avoiding the need to allocate and store revert strings, which also reduces deployment costs. Additionally, custom errors can be defined and used across contracts, interfaces, and libraries.

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

Snapshot:

Solidity Copy

```
bqBTC.sol
  46:             require(
  50:             require(
  56:         require(nativeSent || btcSent, "Insufficient tokens sent to mint");
  61:         require(
  107:         require(
  111:         require(
  124:         require(

Gov.sol
  142:         require(
  146:         require(lpContract.poolActive(params.poolId), "Pool does not exist")
  147:         require(params.claimAmount > 0, "Claim amount must be greater than 0
  203:         require(!voters[_proposalId][msg.sender].voted, "Already voted");
  205:         require(proposal.createdAt != 0, "Proposal does not exist");
  206:         require(
  222:         require(voterWeight > 0, "No voting weight");
  298:         require(
  307:         require(_newDuration > 0, "Voting duration must be greater than 0");
  413:         require(coverContract == address(0), "Governance already set");
  414:         require(
  423:         require(numberofTokens > 0);
  432:         require(isAdmin[msg.sender], "Not authorized");

InsuranceCover.sol
  470:         require(cover.capacity > 0, "Invalid cover capacity");
  557:         require(msg.sender == governance, "Not authorized");
  562:         require(msg.sender == lpAddress, "Not authorized");

InsurancePool.sol
  176:         require(pools[_poolId].isActive, "Pool does not exist or is inactive
  177:         require(_apy > 0, "Invalid APY");
  178:         require(_minPeriod > 0, "Invalid minimum period");
  323:         require(userDeposit.amount > 0, "No deposit found for this address")
  324:         require(userDeposit.status == Status.Active, "Deposit is not active'
  325:         require(
  345:         require(_amount > 0, "Amount must be greater than 0");
  346:         require(selectedPool.isActive, "Pool is inactive or does not exist")
  407:         require(
  412:         require(msg.sender == proposalParam.user, "Not a valid proposal");
  413:         require(pool.isActive, "Pool is not active");
  414:         require(
  506:         require(governance == address(0), "Governance already set");
  507:         require(_governance != address(0), "Governance address cannot be zer
  513:         require(coverContract == address(0), "Governance already set");
  514:         require(
  523:         require(
  531:         require(
```

Recommended Mitigation Steps:

Consider replacing all revert strings with custom errors in the solution, and particularly those that have multiple occurrences:

```
// Define Custom Error
error InsufficientTokens();

// Original code
require(nativeSent || btcSent, "Insufficient tokens sent to mint");

// Refactored code
if (!(nativeSent || btcSent)) revert InsufficientTokens();
```

G-03

Type	Use ++i Instead of i++ or i += 1 for Gas Efficiency
Severity	Gas Error
File	Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	151, 208, 236
Status	**Acknowledged**

Description

In Solidity, using pre-increment (++i) and pre-decrement (--i) operators can be more gas-efficient than post-increment (i++) or addition assignments (i += 1). This is because the compiler can optimize the pre-increment operations without needing to create a temporary variable for returning the incremented value. Utilizing pre-increments and pre-decrements where applicable can save approximately 5 gas per instance.

Snapshot:

```
Gov.sol
  151:         for (uint256 i = 1; i < proposalCounter; i++) {
  180:         for (uint i = 0; i < participants.length; i++) {
  239:         for (uint i = 0; i < participants.length; i++) {
  255:         for (uint256 i = 0; i < proposalIds.length; i++) {
  330:         for (uint256 i = 0; i < proposalIds.length; i++) {
  345:         for (uint256 i = 0; i < proposalIds.length; i++) {
  356:         for (uint256 i = 0; i < proposalIds.length; i++) {
  381:         for (uint256 i = 0; i < proposalIds.length; i++) {
  391:         for (uint256 i = 0; i < proposalIds.length; i++) {

InsuranceCover.sol
  208:         for (uint256 i = 0; i < coversInPool.length; i++) {
  269:         for (uint256 i = 0; i < coversInPool.length; i++) {
  357:         for (uint i = 0; i < participants.length; i++) {
  378:         for (uint256 i = 0; i < coverIds.length; i++) {
  389:         for (uint256 i = 0; i < coverIds.length; i++) {
  406:         for (uint256 i = 0; i < coverIds.length; i++) {
  418:         for (uint256 i = 0; i < coverIds.length; i++) {
  452:         for (uint256 i = 1; i < coverIds.length; i++) {

InsurancePool.sol
  236:         for (uint256 i = 1; i <= poolCount; i++) {
  257:         for (uint i = 0; i < poolToCovers[_poolId].length; i++) {
  282:         for (uint256 i = 1; i <= poolCount; i++) {
  293:         for (uint256 i = 1; i <= poolCount; i++) {
  333:         for (uint i = 0; i < poolCovers.length; i++) {
  381:         for (uint i = 0; i < poolCovers.length; i++) {
  386:         for (uint i = 0; i < participants.length; i++) {
  424:         for (uint i = 0; i < poolCovers.length; i++) {
```

Recommended Mitigation Steps:

To optimize gas usage, follow these steps:

```
// Original code
for (uint256 i = 1; i < proposalCounter; i++) { ... }

// Refactored code
for (uint256 i = 1; i < proposalCounter; ++i) { ... }
```

G-04

Type	Splitting require() statements that use && saves gas
Severity	Gas Error
File	bqBTC.sol
Instances	108
Status	**Acknowledged**

Description

In Solidity, combining multiple conditions in a single require() statement using the logical AND operator (&&) can lead to increased gas costs. This is because if any condition fails, the entire expression must be evaluated, which can be avoided by splitting the conditions into separate require() statements. Each require() statement will stop execution as soon as the first condition fails, potentially saving gas.

Snapshot:

```
bqBTC.sol
  108:             pool != address(0) && cover != address(0) && gov != a
  108:             pool != address(0) && cover != address(0) && gov != a
  112:             poolAddress == address(0) &&
  113:                 coverAddress == address(0) &&
```

Recommended Mitigation Steps:

To optimize gas usage, follow these steps:

G-05

Type	Increments/decrements can be unchecked in for-loops
Severity	Gas Error
File	Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	151, 208, 236
Status	**Acknowledged**

Description

In Solidity 0.8 and later, the compiler includes default overflow checks for unsigned integers. While this feature enhances security, it can also lead to increased gas costs during iterations in loops. Developers can optimize gas usage by using the unchecked block for increments and decrements in for-loops, which saves approximately 25 gas per instance.
The same can be applied with decrements (which should use break when i == 0). The risk of overflow is non-existent for uint256.

ethereum/solidity#10695

Proof of Concept:

```
Gov.sol
  151:          for (uint256 i = 1; i < proposalCounter; i++) {
  255:          for (uint256 i = 0; i < proposalIds.length; i++) {
  276:         for (uint256 j = 0; j < correctVoters.length; j++) {
  285:         for (uint256 j = 0; j < correctVoters.length; j++) {
  330:          for (uint256 i = 0; i < proposalIds.length; i++) {
  345:          for (uint256 i = 0; i < proposalIds.length; i++) {
  356:          for (uint256 i = 0; i < proposalIds.length; i++) {
  381:          for (uint256 i = 0; i < proposalIds.length; i++) {
  391:          for (uint256 i = 0; i < proposalIds.length; i++) {

InsuranceCover.sol
  208:          for (uint256 i = 0; i < coversInPool.length; i++) {
  269:          for (uint256 i = 0; i < coversInPool.length; i++) {

  378:          for (uint256 i = 0; i < coverIds.length; i++) {
  389:          for (uint256 i = 0; i < coverIds.length; i++) {
  406:          for (uint256 i = 0; i < coverIds.length; i++) {
  418:          for (uint256 i = 0; i < coverIds.length; i++) {
  452:          for (uint256 i = 1; i < coverIds.length; i++) {

InsurancePool.sol
  236:          for (uint256 i = 1; i <= poolCount; i++) {
  282:          for (uint256 i = 1; i <= poolCount; i++) {
  293:          for (uint256 i = 1; i <= poolCount; i++) {
```

Recommended Mitigation Steps:

To optimize gas usage by utilizing unchecked increments and decrements in loops, follow these steps:

```
// Original code
for (uint256 i = 1; i < proposalCounter; i++) { ... }

// Refactored code
for (uint256 i = 1;) {
    if (i >= proposalCounter) break;
    // Loop logic...
    unchecked { ++i; }
}
```

G-06

Type	Use != 0 instead of > 0 for unsigned integer comparison
Severity	Gas Error
File	Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	147, 380, 177
Status	**Acknowledged**

Description

In Solidity, when checking if an unsigned integer is non-zero, it is more efficient and clearer to use != 0 instead of > 0. This change can enhance readability and potentially save gas, as the comparison is straightforward and directly conveys the intent of checking for a non-zero value.

Snapshot:

```
Gov.sol
  147:         require(params.claimAmount > 0, "Claim amount must be gre
  222:         require(voterWeight > 0, "No voting weight");
  307:         require(_newDuration > 0, "Voting duration must be greate
  423:         require(numberofTokens > 0);

InsuranceCover.sol
  380:             if (userCovers[user][id].coverValue > 0) {
  391:             if (userCovers[user][id].coverValue > 0) {
  470:         require(cover.capacity > 0, "Invalid cover capacity");

InsurancePool.sol
  177:         require(_apy > 0, "Invalid APY");
  178:         require(_minPeriod > 0, "Invalid minimum period");
  284:             if (pool.deposits[_userAddress].amount > 0) {
  301:             if (pool.deposits[_userAddress].amount > 0) {
  323:         require(userDeposit.amount > 0, "No deposit found for thi
  345:         require(_amount > 0, "Amount must be greater than 0");
  351:         if (selectedPool.deposits[msg.sender].amount > 0) {
```

Recommended Mitigation Steps:

To optimize gas usage, follow these steps:

```
// Original code
require(params.claimAmount > 0, "Claim amount must be greater than
0");

// Refactored code
require(params.claimAmount != 0, "Claim amount must be greater than
0");
```

I-01

Type	Use a modifier instead of a require/if statement for a special msg.sender actor
Severity	Informational
File	bqBTC.sol, Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	62, 299, 557
Status	**Acknowledged**

Description

When implementing access control in Solidity smart contracts, using a modifier is preferred over inline require or if statements. This approach enhances code readability and maintainability by clearly defining the access control logic in a reusable manner.

Snapshot:

```
bqBTC.sol
  62:             msg.sender == initialOwner ||
  63:                 msg.sender == poolAddress ||
  64:                 msg.sender == coverAddress,
  125:             msg.sender == coverAddress ||
  126:                 msg.sender == initialOwner ||
  127:                 msg.sender == govContract ||
  128:                 msg.sender == poolAddress,

Gov.sol
  299:             msg.sender == proposals[proposalId].proposalParam.user ||
  300:                 msg.sender == poolContract,

InsuranceCover.sol
  557:         require(msg.sender == governance, "Not authorized");
  562:         require(msg.sender == lpAddress, "Not authorized");

InsurancePool.sol
  412:         require(msg.sender == proposalParam.user, "Not a valid proposal");
  524:             msg.sender == governance || msg.sender == initialOwner,
  524:             msg.sender == governance || msg.sender == initialOwner,
  532:             msg.sender == coverContract || msg.sender == initialOwner,
  532:             msg.sender == coverContract || msg.sender == initialOwner,
```

Recommended Mitigation Steps:

Review the identified lines of code where msg.sender checks are used for access control in multiple locations.
Define reusable modifiers that encapsulate the access control logic. Replace the existing require statements and inline checks with the newly created modifiers in your functions

I-02

Type	Variables need not be initialized to zero
Severity	Informational
File	Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	180, 208, 257
Status	**Acknowledged**

Description

In Solidity, the default value for uninitialized state variables is zero. Therefore, explicitly initializing these variables to zero is unnecessary and can lead to code that is less readable and more cluttered. This report identifies instances in the provided smart contracts where loop variables are initialized to zero unnecessarily.

```
Gov.sol
  180:          for (uint i = 0; i < participants.length; i++) {
  239:          for (uint i = 0; i < participants.length; i++) {
  255:          for (uint256 i = 0; i < proposalIds.length; i++) {
  330:          for (uint256 i = 0; i < proposalIds.length; i++) {
  345:          for (uint256 i = 0; i < proposalIds.length; i++) {
  356:          for (uint256 i = 0; i < proposalIds.length; i++) {
  381:          for (uint256 i = 0; i < proposalIds.length; i++) {
  391:          for (uint256 i = 0; i < proposalIds.length; i++) {

InsuranceCover.sol
  208:          for (uint256 i = 0; i < coversInPool.length; i++) {
  269:          for (uint256 i = 0; i < coversInPool.length; i++) {
  357:          for (uint i = 0; i < participants.length; i++) {
  378:          for (uint256 i = 0; i < coverIds.length; i++) {
  389:          for (uint256 i = 0; i < coverIds.length; i++) {
  406:          for (uint256 i = 0; i < coverIds.length; i++) {
  418:          for (uint256 i = 0; i < coverIds.length; i++) {

InsurancePool.sol
  257:          for (uint i = 0; i < poolToCovers[_poolId].length; i++)
  333:          for (uint i = 0; i < poolCovers.length; i++) {
  381:          for (uint i = 0; i < poolCovers.length; i++) {
  386:          for (uint i = 0; i < participants.length; i++) {
  424:          for (uint i = 0; i < poolCovers.length; i++) {
```

Recommended Mitigation Steps:

Instead of initializing loop variables with uint x = 0, simply declare them using uint x; or uint256 x; based on your needs.

I-03

Type	Missing NatSpec
Severity	Informational
File	Gov.sol, InsuranceCover.sol, InsurancePool.sol
Instances	-------
Status	**Acknowledged**

Description

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables, and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec). Most of the functions have missing Ethereum Natural Specification Format (NatSpec) comments.

Proof of Concept:

Multiple functions across InsuranceCover.sol, InsurancePool.sol and Gov.sol contract lacks NatSpec documentation for several functions.

Recommended Mitigation:

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).

APPENDIX

Auditing Approach and Methodologies applied

The Block Audit Report team has performed rigorous testing of the project including the analysis of the code design patterns where we reviewed the smart contract architecture to ensure it is structured along with the safe use of standard inherited contracts and libraries. Our team also conducted a formal line by line inspection of the Smart Contract i.e., a manual review, to find potential issues including but not limited to

- Race conditions
- Zero race conditions approval attacks
- Re-entrancy
- Transaction-ordering dependence
- Timestamp dependence
- Check-effects-interaction pattern (optimistic accounting)
- Decentralized denial-of-service attacks
- Secure ether transfer pattern
- Guard check pattern
- Fail-safe mode
- Gas-limits and infinite loops
- Call Stack depth

In the Unit testing Phase, we coded/conducted custom unit tests written against each function in the contract to verify the claimed functionality from our client. In Automated Testing, we tested the Smart Contract with our standard set of multifunctional tools to identify vulnerabilities and security flaws. The code was tested in collaboration of our multiple team members and this included but not limited to;

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.
- Analyzing the complexity of the code in depth and in detail line-by-line manual review of the code.
- Deploying the code on testnet using multiple clients to run live tests.
- Analyzing 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.
- Analyzing the security of the on-chain data.

Issue Categories:

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

Critical Severity Issues

Issues of Critical Severity leaves smart contracts vulnerable to major exploits and can lead to asset loss and data loss. These can have significant impact on the functionality/performance of the smart contract.
We recommend these issues must be fixed before proceeding to MainNet..

High Severity Issues

Issues of High Severity are not as easy to exploit but they might endanger the execution of the smart contract and potentially create crucial problems.
Fixing these issues is highly recommended before proceeding to MainNet.

Medium Severity Issues

Issues on this level are not a major cause of vulnerability to the smart contract, they cannot lead to data-manipulations or asset loss but may affect funtionality.
It is important to fix these issues before proceeding to MainNet.

Low Severity Issues

Issues at this level are very low in their impact on the overall functionality and execution of the smart contract. These are mostly code-level violations or improper formatting.
These issues can be remain unfixed or can be fixed at a later date if the code is redeployed or forked.

Informational Findings

These are finding that our team comes accross when manually reviewing a smart contract which are important to know for the owners as well as users of a contract.
These issues must be acknowledged by the owners before we publish our report.

Ownership Priviledges

Owner of a smart contract can include certain rights and priviledges while deploying a smart contract that might be hidden deep inside the codebase and may make the project vulnerable to rug-pulls or other types of scams.
We at BlockAudit believe in transparency and hence we showcase Ownership priviledges separately so the owner as well as the investors can get a better understanding about the project.

DISCLAIMER

This is a limited report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for the client to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that the client should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for the client to conduct the client's own independent investigations before making any decisions. We go into more detail on this in the below disclaimer below – please make sure to read it in full.

By reading this report or any part of it, the client agrees to the terms of this disclaimer. If the client does not agree to the terms, then please immediately cease reading this report, and delete and destroy any/all copies of this report downloaded and/or printed by the client. This report is provided for information purposes only and stays on a non-reliance basis, and does not constitute investment advice. No one/ NONE shall have any rights to rely on the report or its contents, and BlockAudit and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives).

(BlockAudit) owes no duty of care towards the client or any other person, nor does BlockAudit claim any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and BlockAudit hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effects in relation to the report.

Except and only to the extent that it is prohibited by law, BlockAudit hereby excludes all liability and responsibility, and neither the client nor any other person shall have any claim against BlockAudit, for any amount or kind of loss or damage that may result to the client or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report.

The analysis of the security is purely based on the received smart contracts alone. No related/third-party smart contracts, applications or operations were reviewed for security. No product code has been reviewed.

Note: The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the **BQ Labs** team put a bug bounty program in place to encourage further analysis of the smart contracts by other third parties

About BlockAudit

BlockAudit is an industry leading security organisation that helps web3 blockchain based projects with their security and correctness of their smart-contracts. With years of experience we have a dedicated team that is capable of performing audits in a wide variety of languages including HTML, PHP, JS, Node, React, Native, Solidity, Rust and other Web3 frameworks for DApps, DeFi, GameFi and Metaverse platforms.

With a mission to make web3 a safe and secure place BlockAudit is commited to provide it's partners with a budget and investor friendly security Audit Report that will increase the value of their projects significantly.

 *www.blockaudit.report*

 *team@blockaudit.report*

 *@BlockAudit*

 *github.com/Block-Audit-Report*