# BlocksSpace

## Smart Contract Audit Report
## Prepared for 1000Blocks


| | |
|---|---|
| **Date Issued:** | Sep 17, 2021 |
| **Project ID:** | AUDIT2021012 |
| **Version:** | v3.0 |
| **Confidentiality Level:** | Public |

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

## Report Information

| | |
|---|---|
| **Project ID** | AUDIT2021012 |
| **Version** | v3.0 |
| **Client** | 1000Blocks |
| **Project** | BlocksSpace |
| **Auditor(s)** | Weerawat Pawanawiwat<br>Suvicha Buakhom<br>Patipon Suwanbol<br>Peeraphut Punsuwan |
| **Author** | Weerawat Pawanawiwat |
| **Reviewer** | Pongsakorn Sommalai |
| **Confidentiality Level** | Public |

## Version History

| Version | Date | Description | Author(s) |
|---|---|---|---|
| 3.0 | Sep 17, 2021 | Update issues' statuses | Patipon Suwanbol |
| 2.0 | Aug 25, 2021 | Update executive summary | Weerawat Pawanawiwat |
| 1.0 | Aug 23, 2021 | Full report | Weerawat Pawanawiwat |

## Contact Information



| | |
|---|---|
| **Company** | Inspex |
| **Phone** | (+66) 90 888 7186 |
| **Telegram** | t.me/inspexco |
| **Email** | audit@inspex.co |

# Table of Contents

# 1. Executive Summary

As requested by 1000Blocks, Inspex team conducted an audit to verify the security posture of the BlocksSpace smart contracts between Aug 10, 2021 and Aug 11, 2021. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of BlocksSpace smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

Major parts of the BlocksSpace smart contracts are custom-made, written by the 1000Blocks team, and some parts adapted the traditional MasterChef design to extend the functionalities for the rewards distribution. Since the design is newly implemented and not from the old designs heavily used or tested in multiple platforms, most issues found are related to the reward distribution mechanism, causing the rewards to be inaccurately calculated.

Designing new usabilities for smart contracts is great for the blockchain ecosystem, opening it up for more applications and adaptations in the future. We hope that this audit can leverage the security level of the BlocksSpace smart contracts without compromising on the creativity and business usability of the platform.

## 1.1. Audit Result

In the initial audit, Inspex found 4 high, 4 medium, 1 low, 1 very low, and 3 info-severity issues. With the project team's prompt response, 4 high, 4 medium, 1 low, 1 very low, and 1 info-severity issues were resolved or mitigated in the reassessment, while 2 info-severity issues were acknowledged by the team. Therefore, Inspex trusts that BlocksSpace smart contracts have sufficient protections to be safe for public use.


## 1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one

single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

# 2. Project Overview

## 2.1. Project Introduction

1000Blocks Space is a project where users create community-powered NFTs that yield returns in the process of NFT creation to make digital art creation fun and rewarding.

BlocksSpace is the core decentralized application of 1000Blocks that allows users to own block areas in the available spaces. The users can put images in their own areas, and other users can pay a higher price to take over the occupied block areas. The owner of the block areas can gain $BLS tokens as rewards, and $BLS tokens can also be used to stake for gaining $BNB that is collected from the block area costs.

**Scope Information:**

| | |
|---|---|
| **Project Name** | BlocksSpace |
| **Website** | https://app.1000blocks.space/spaces |
| **Smart Contract Type** | Ethereum Smart Contract |
| **Chain** | Binance Smart Chain |
| **Programming Language** | Solidity |

**Audit Information:**

| | |
|---|---|
| **Audit Method** | Whitebox |
| **Audit Date** | Aug 10, 2021 - Aug 11, 2021 |
| **Reassessment Date** | Aug 20, 2021 |

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

## 2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

**Initial Audit: (Commit: 51efffaa45e95db5e28c5d550d351b84a03d098f)**

| Contract | Location (URL) |
| --- | --- |
| BLSToken | https://github.com/1000Blocks-space/smart-contracts/blob/51efffaa45/hardhat/contracts/BLSToken.sol |
| BlocksRewardsManager | https://github.com/1000Blocks-space/smart-contracts/blob/51efffaa45/hardhat/contracts/BlocksRewardsManager.sol |
| BlocksSpace | https://github.com/1000Blocks-space/smart-contracts/blob/51efffaa45/hardhat/contracts/BlocksSpace.sol |
| BlocksStaking | https://github.com/1000Blocks-space/smart-contracts/blob/51efffaa45/hardhat/contracts/BlocksStaking.sol |

**Reassessment: (Commit: 8311a0436dba5f168fe830bd84ffc2832f8a1b38)**

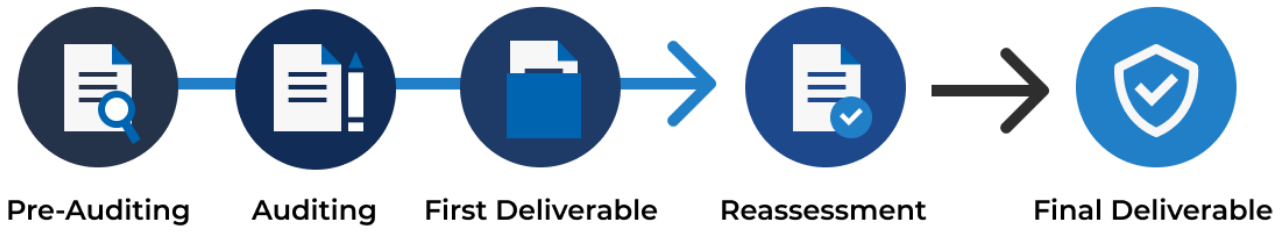
| Contract | Location (URL) |
| --- | --- |
| BLSToken | https://github.com/1000Blocks-space/smart-contracts/blob/8311a0436d/hardhat/contracts/BLSToken.sol |
| BlocksRewardsManager | https://github.com/1000Blocks-space/smart-contracts/blob/8311a0436d/hardhat/contracts/BlocksRewardsManager.sol |
| BlocksSpace | https://github.com/1000Blocks-space/smart-contracts/blob/8311a0436d/hardhat/contracts/BlocksSpace.sol |
| BlocksStaking | https://github.com/1000Blocks-space/smart-contracts/blob/8311a0436d/hardhat/contracts/BlocksStaking.sol |

The assessment scope covers only the in-scope smart contracts and the smart contracts that they are inherited from.

# 3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue




## 3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

## 3.2. Audit Items

The following audit items were checked during the auditing activity.

| General |
| --- |
| Reentrancy Attack |
| Integer Overflows and Underflows |
| Unchecked Return Values for Low-Level Calls |
| Bad Randomness |
| Transaction Ordering Dependence |
| Time Manipulation |
| Short Address Attack |
| Outdated Compiler Version |
| Use of Known Vulnerable Component |
| Deprecated Solidity Features |
| Use of Deprecated Component |
| Loop with High Gas Consumption |
| Unauthorized Self-destruct |
| Redundant Fallback Function |
| **Advanced** |
| Business Logic Flaw |
| Ownership Takeover |
| Broken Access Control |
| Broken Authentication |
| Upgradable Without Timelock |
| Improper Kill-Switch Mechanism |
| Improper Front-end Integration |
| Insecure Smart Contract Initiation |

| |
|---|
| Denial of Service |
| Improper Oracle Usage |
| Memory Corruption |
| **Best Practice** |
| Use of Variadic Byte Array |
| Implicit Compiler Version |
| Implicit Visibility Level |
| Implicit Type Inference |
| Function Declaration Inconsistency |
| Token API Violation |
| Best Practices Violation |

## 3.3. Risk Rating

OWASP Risk Rating Methodology[1] is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker.
- **Impact**: a measure of the damage caused by a successful attack

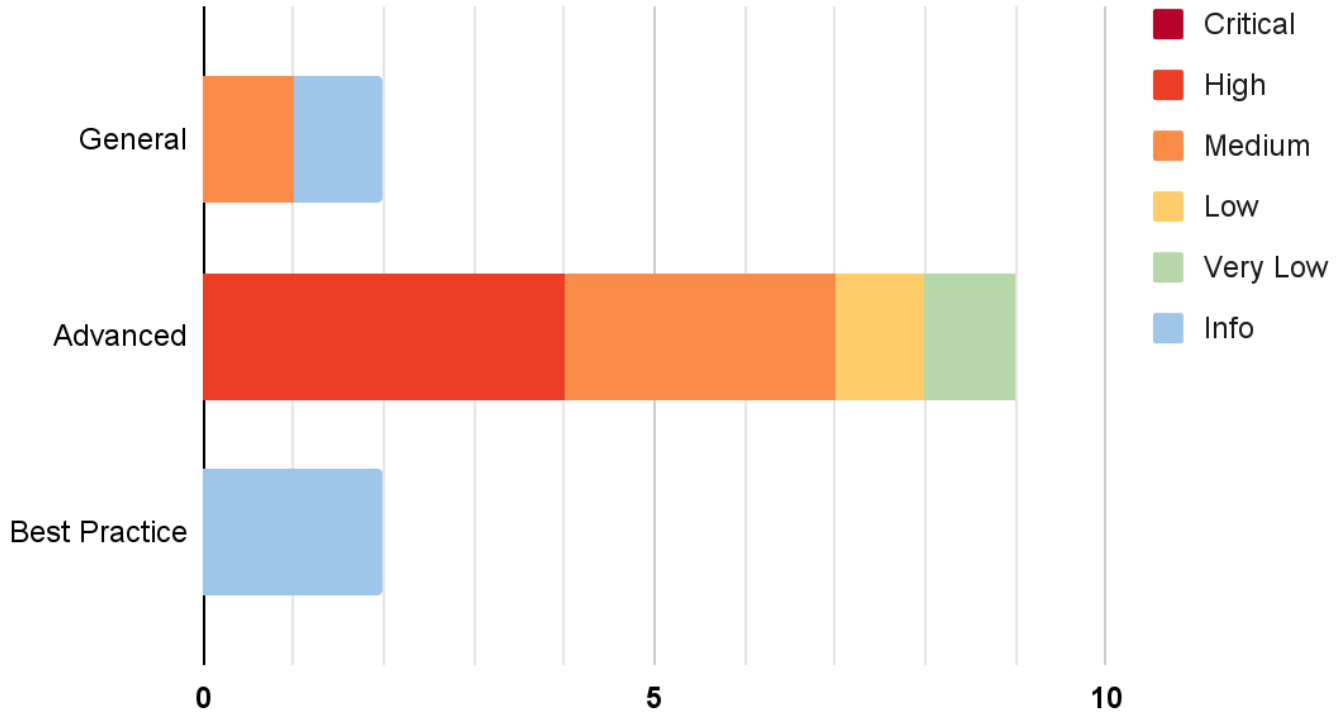
Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

**Severity** is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

| Impact \ Likelihood | Low | Medium | High |
|---|---|---|---|
| **Low** | **Very Low** | **Low** | **Medium** |
| **Medium** | **Low** | **Medium** | **High** |
| **High** | **Medium** | **High** | **Critical** |

# 4. Summary of Findings

From the assessments, Inspex has found 13 issues in three categories. The following chart shows the number of the issues categorized into three categories: **General**, **Advanced**, and **Best Practice**.




The statuses of the issues are defined as follows:

| Status | Description |
|---|---|
| Resolved | The issue has been resolved and has no further complications. |
| Resolved * | The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5. |
| Acknowledged | The issue's risk has been acknowledged and accepted. |
| No Security Impact | The best practice recommendation has been acknowledged. |

The information and status of each issue can be found in the following table:

| ID | Title | Category | Severity | Status |
|---|---|---|---|---|
| IDX-001 | Arbitrary Share Amount Setting | Advanced | **High** | Resolved |
| IDX-002 | Incorrect Reward Calculation from takeoverRewards | Advanced | **High** | Resolved |
| IDX-003 | Incorrect Reward Calculation from allUsersRewardDebt | Advanced | **High** | Resolved |
| IDX-004 | Incorrect Reward Calculation from Total Rewards Rate | Advanced | **High** | Resolved |
| IDX-005 | Token Stealing via BlocksRewardsManager Address Setting | Advanced | **Medium** | Resolved |
| IDX-006 | Token Stealing via BlocksStaking Address Setting | Advanced | **Medium** | Resolved * |
| IDX-007 | Centralized Control of State Variable | General | **Medium** | Resolved |
| IDX-008 | Incorrect Condition | Advanced | **Medium** | Resolved |
| IDX-009 | Incorrect Reward Calculation from blsPerBlockAreaPerBlock | Advanced | **Low** | Resolved |
| IDX-010 | Insufficient Logging for Privileged Functions | Advanced | **Very Low** | Resolved |
| IDX-011 | Outdated Compiler Version | Best Practice | **Info** | No Security Impact |
| IDX-012 | Improper Function Visibility | Best Practice | **Info** | No Security Impact |
| IDX-013 | Inexplicit Solidity Compiler Version | Best Practice | **Info** | Resolved |

* The mitigations or clarifications by 1000Blocks can be found in Chapter 5.

# 5. Detailed Findings Information

## 5.1. Arbitrary Share Amount Setting

| | |
|---|---|
| **ID** | IDX-001 |
| **Target** | BlocksRewardsManager |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-284: Improper Access Control |
| **Risk** | **Severity:** High<br><br>**Impact:** High<br>The contract owner can increase the share amount of any address on any space. With a high number of shares, the owner will be able to claim most of the $BLS rewards distributed from `BlocksRewardsManager`.<br><br>**Likelihood:** Medium<br>There is no restriction to prevent the owner from performing this attack. |
| **Status** | Resolved<br>The 1000Blocks team has resolved this issue as suggested in commit `c07e953313e13f1f3ba89238bef639a899817470` by checking the `spaceId` from the caller address. |

### 5.1.1. Description

In the `BlocksRewardsManager` contract, the `onlySpace` modifier is an access control modifier that allows only the whitelisted addresses to execute the functions with this modifier. The address is checked using the value stored in the `spacesByAddress` mapping.

**BlocksRewardsManager.sol**

```
50 modifier onlySpace() {
51     require(spacesByAddress[msg.sender] == true, "Not a space.");
52     _;
53 }
```

The entries in `spacesByAddress` mapping can be set to `true` using the `addSpace()` function in line 70, which can be executed by the owner.

**BlocksRewardsManager.sol**

```
69 function addSpace(address spaceContract_, uint256 blsPerBlockAreaPerBlock_)
   external onlyOwner {
70     spacesByAddress[spaceContract_] = true;
71     uint256 spaceId = spaceInfo.length;
```

```
72      SpaceInfo storage newSpace = spaceInfo.push();
73      newSpace.contractAddress = spaceContract_;
74      newSpace.spaceId = spaceId;
75      newSpace.blsPerBlockAreaPerBlock = blsPerBlockAreaPerBlock_;
76      allBlsPerBlockAreaPerBlock = allBlsPerBlockAreaPerBlock +
    blsPerBlockAreaPerBlock_;
77      emit SpaceAdded(spaceId, spaceContract_, msg.sender);
78  }
```

The `onlySpace` modifier is used in the `blocksAreaBoughtOnSpace()` function, and since the contract owner can add the owner's addresses to the whitelist, the owner can freely execute this function.

Therefore, the owner can set the `previousBlockOwners_` parameter to be an array with a large number of elements. As a result, the `numberOfBlocksBought` variable will be set to a large number in line 123, and the amount of shares will be increased by in line 125.

**BlocksRewardsManager.sol**

```
107 function blocksAreaBoughtOnSpace(
108     uint256 spaceId_,
109     address buyer_,
110     address[] calldata previousBlockOwners_,
111     uint256[] calldata previousOwnersPrices_
112 ) public payable onlySpace {
113     SpaceInfo storage space = spaceInfo[spaceId_];
114     UserInfo storage user = userInfo[spaceId_][buyer_];
115     uint256 blsPerBlockAreaPerBlock = space.blsPerBlockAreaPerBlock;
116 
117     // If user already had some block.areas then calculate all rewards pending
118     if (user.lastRewardCalculatedBlock > 0) {
119         uint256 multiplier = getMultiplier(user.lastRewardCalculatedBlock);
120         uint256 blsRewards = multiplier * blsPerBlockAreaPerBlock;
121         user.pendingRewards = user.pendingRewards + user.amount * blsRewards;
122     }
123     uint256 numberOfBlocksBought = previousBlockOwners_.length;
124     // Set user data
125     user.amount = user.amount + numberOfBlocksBought;
126     user.lastRewardCalculatedBlock = block.number;
```

And since the amount of shares is used in the $BLS reward calculation in line 221 and 224-228, the owner can get a very high amount of rewards from the increase of share amount.

**BlocksRewardsManager.sol**

```
219 function claim(uint256 spaceId_) public {
220     UserInfo storage user = userInfo[spaceId_][msg.sender];
221     uint256 amount = user.amount;
```

```
222     uint256 lastRewardCalculatedBlock = user.lastRewardCalculatedBlock;
223     if (amount > 0 && lastRewardCalculatedBlock < block.number) {
224         user.pendingRewards =
225             user.pendingRewards +
226             amount *
227             getMultiplier(lastRewardCalculatedBlock) *
228             spaceInfo[spaceId_].blsPerBlockAreaPerBlock;
229         user.lastRewardCalculatedBlock = block.number;
230     }
231     uint256 toClaimAmount = user.pendingRewards;
232     if (toClaimAmount > 0) {
233         uint256 claimedAmount = safeBlsTransfer(msg.sender, toClaimAmount);
234         emit Claim(msg.sender, claimedAmount);
235         // This is also kinda check, since if user claims more than eligible,
    this will revert
236         user.pendingRewards = toClaimAmount - claimedAmount;
237         blsRewardsClaimed = blsRewardsClaimed + claimedAmount; // Globally
    claimed rewards, for proper end distribution calc
238     }
239 }
```

### 5.1.2. Remediation

Inspex suggests removing the `onlySpace` modifier and using the contract address to determine the `spaceId` to prevent the owner from being able to set the user amount in any space, for example:

First, the `spaceIdMapping` mapping to map the contract address to `spaceId` should be created.

**BlocksRewardsManager.sol**

```
33    address payable public treasury;
34     IERC20 public blsToken;
35     BlocksStaking public blocksStaking;
36     SpaceInfo[] public spaceInfo;
37     mapping(uint256 => mapping(address => UserInfo)) public userInfo;
38     mapping(address => uint256) public spaceIdMapping;
```

Then, we suggest adding the space address to the `spaceIdMapping` mapping in the `addSpace()` function.

**BlocksRewardsManager.sol**

```
69  function addSpace(address spaceContract_, uint256 blsPerBlockAreaPerBlock_)
    external onlyOwner {
70         require(spaceIdMapping[spaceContract_] == 0, "Space is already
    added.");
71         uint256 spaceId = spaceInfo.length;
72         spaceIdMapping[spaceContract_] = spaceId;
73         SpaceInfo storage newSpace = spaceInfo.push();
```

```
74          newSpace.contractAddress = spaceContract_;
75          newSpace.spaceId = spaceId;
76          newSpace.blsPerBlockAreaPerBlock = blsPerBlockAreaPerBlock_;
77          allBlsPerBlockAreaPerBlock = allBlsPerBlockAreaPerBlock +
    blsPerBlockAreaPerBlock_;
78          emit SpaceAdded(spaceId, spaceContract_, msg.sender);
79      }
```

Finally, it is recommended to remove the `spaceId_` parameter, remove the `onlySpace` modifier, and get the value of `spaceId_` from the address of the caller using the mapping.

**BlocksRewardsManager.sol**

```
107     function blocksAreaBoughtOnSpace(
108         address buyer_,
109         address[] calldata previousBlockOwners_,
110         uint256[] calldata previousOwnersPrices_
111     ) public payable {
112         address memory spaceId_ = spaceIdMapping[msg.sender];
113         SpaceInfo storage space = spaceInfo[spaceId_];
114         require(space.contractAddress == msg.sender, "Not a space contract");
115         UserInfo storage user = userInfo[spaceId_][buyer_];
116         uint256 blsPerBlockAreaPerBlock = space.blsPerBlockAreaPerBlock;
```

## 5.2. Incorrect Reward Calculation from takeoverRewards

| | |
|---|---|
| **ID** | IDX-002 |
| **Target** | BlocksStaking |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-840: Business Logic Errors |
| **Risk** | **Severity:** High<br><br>**Impact:** Medium<br>The miscalculation can cause the users to gain less rewards, and some $BNB rewards to be stuck in the contract and unclaimable by the users.<br><br>**Likelihood:** High<br>The miscalculation can occur whenever block areas are taken over. |
| **Status** | Resolved<br>The 1000Blocks team has resolved this issue as suggested in commit `bb16d18f74037f9264c6b9d45a79d883a1958988` by deducting the `takeoverRewards` in the `claim()` function. |

### 5.2.1. Description

In the `BlocksStaking` contract, the `takeoverRewards` state variable stores the total amount of $BNB assigned to the previous owners of block areas that are taken over.

**BlocksStaking.sol**

```
153 function distributeRewards(address[] calldata addresses_, uint256[] calldata
    rewards_) public payable {
154     uint256 tmpTakeoverRewards;
155     for (uint256 i = 0; i < addresses_.length; ++i) {
156         // process each reward for covered blocks
157         userInfo[addresses_[i]].takeoverReward =
    userInfo[addresses_[i]].takeoverReward + rewards_[i]; // each user that got
    blocks covered gets a reward
158         tmpTakeoverRewards = tmpTakeoverRewards + rewards_[i];
159     }
160     takeoverRewards = takeoverRewards + tmpTakeoverRewards;
161
162     // what remains is the reward for staked amount
163     if (msg.value - tmpTakeoverRewards > 0 && totalTokens > 0) {
164         // Update rewards per share because balance changes
165         accRewardsPerShare = accRewardsPerShare + (rewardsPerBlock *
    getMultiplier()) / totalTokens;
166         lastRewardCalculatedBlock = block.number;
```

```
167             calculateRewardsDistribution();
168         }
169     }
```

It is used to calculate the leftover amount of $BNB in the contract to be distributed as the rewards.

**BlocksStaking.sol**

```
171 function calculateRewardsDistribution() internal {
172     uint256 allReservedRewards = (accRewardsPerShare * totalTokens) / 1e12;
173     uint256 availableForDistribution = (address(this).balance +
    allUsersRewardDebt - allReservedRewards - takeoverRewards);
174     rewardsPerBlock = (availableForDistribution * 1e12) /
    rewardsDistributionPeriod;
175     rewardsFinishedBlock = block.number + rewardsDistributionPeriod;
176 }
```

However, even when the users have claimed their rewards, the `takeoverRewards` state variable is not deducted.

**BlocksStaking.sol**

```
135 function claim() public {
136     uint256 reward = pendingRewards(msg.sender);
137
138     if (reward <= 0) return; // skip if no rewards
139
140     UserInfo storage user = userInfo[msg.sender];
141     user.rewardDebt = user.rewardDebt + reward; // reset: cache current total
    reward per token
142     user.takeoverReward = 0; // reset takeover reward
143
144     // transfer reward in BNBs to the user
145     (bool success, ) = msg.sender.call{value: reward}("");
146     require(success, "Transfer failed.");
147     emit Claim(msg.sender, reward);
148 }
```

This causes the value of `takeoverRewards` to keep getting higher every time the block areas are taken over, and the `rewardsPerBlock` will be lower than what it should be. Therefore, not all $BNB will be distributed and some will be stuck and unclaimable in the contract.

### 5.2.2. Remediation

Inspex suggests deducting the `takeoverRewards` every time the reward is claimed, for example:

**BlocksStaking.sol**

```
135 function claim() public {
136     uint256 reward = pendingRewards(msg.sender);
137 
138     if (reward <= 0) return; // skip if no rewards
139 
140     UserInfo storage user = userInfo[msg.sender];
141     user.rewardDebt = user.rewardDebt + reward; // reset: cache current total
    reward per token
142     takeoverRewards = takeoverRewards - user.takeoverReward;
143     user.takeoverReward = 0; // reset takeover reward
144 
145     // transfer reward in BNBs to the user
146     (bool success, ) = msg.sender.call{value: reward}("");
147     require(success, "Transfer failed.");
148     emit Claim(msg.sender, reward);
149 }
```

Please note that the remediations for other issues are not yet applied to the example above.

## 5.3. Incorrect Reward Calculation from allUsersRewardDebt

| | |
|---|---|
| **ID** | IDX-003 |
| **Target** | BlocksStaking |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-840: Business Logic Errors |
| **Risk** | **Severity:** High<br><br>**Impact:** Medium<br>The miscalculation can cause the users to gain less rewards, and some $BNB rewards will be stuck in the contract and unclaimable by the users.<br><br>**Likelihood:** High<br>Due to the incorrect logic, the miscalculation will surely happen. |
| **Status** | Resolved<br>The 1000Blocks team has resolved this issue as suggested in the following commits:<br><br>- a2e71a0d062a2e8f314933501329eaf60259d7f9<br>- 36678477ea1bfaaf4f1f948dac88d26fb03df7d0<br>- 8311a0436dba5f168fe830bd84ffc2832f8a1b38 |

### 5.3.1. Description

In the `BlocksStaking` contract, the `allUsersRewardDebt` state variable is used to store the total reward debt of all users. It is one of the factors to calculate the amount of $BNB available to be distributed.

**BlocksStaking.sol**

```
171 function calculateRewardsDistribution() internal {
172     uint256 allReservedRewards = (accRewardsPerShare * totalTokens) / 1e12;
173     uint256 availableForDistribution = (address(this).balance +
    allUsersRewardDebt - allReservedRewards - takeoverRewards);
174     rewardsPerBlock = (availableForDistribution * 1e12) /
    rewardsDistributionPeriod;
175     rewardsFinishedBlock = block.number + rewardsDistributionPeriod;
176 }
```

However, the `allUsersRewardDebt` is updated in the `deposit()` function only, using the share amount of just the latest user in line 104, not all users, causing the value to be less than what it should be.

**BlocksStaking.sol**

```
86 function deposit(uint256 amount_) public {
87     UserInfo storage user = userInfo[msg.sender];
```

```
88      // if there are staked amount, fully harvest current reward
89      if (user.amount > 0) {
90          claim();
91      }
92
93      if (totalTokens > 0) {
94          accRewardsPerShare = accRewardsPerShare + (rewardsPerBlock *
    getMultiplier()) / totalTokens;
95      } else {
96          calculateRewardsDistribution(); // Means first time any user deposits,
    so start distributing
97      }
98
99      lastRewardCalculatedBlock = block.number;
100
101     totalTokens = totalTokens + amount_; // sum of total staked amount
102     user.amount = user.amount + amount_; // cache staked amount count for this
    wallet
103     user.rewardDebt = (accRewardsPerShare * user.amount) / 1e12; // cache
    current total reward per token
104     allUsersRewardDebt = (accRewardsPerShare * user.amount) / 1e12;
105     emit Deposit(msg.sender, amount_);
106     // Transfer BLS amount from the user to this contract
107     blsToken.safeTransferFrom(address(msg.sender), address(this), amount_);
108 }
```

Since the `allUsersRewardDebt` state is incorrectly updated, the miscalculation of `availableForDistribution` can cause the users to gain less rewards, and some $BNB rewards will be stuck in the contract and unclaimable by the users.

### 5.3.2. Remediation

Inspex suggests fixing the updating of `allUsersRewardDebt` state to be the real accumulation of all users' reward debt, for example:

First, we suggest updating the `allUsersRewardDebt` by using the difference between the user's old and new reward debt.

**BlocksStaking.sol**

```
86  function deposit(uint256 amount_) public {
87          UserInfo storage user = userInfo[msg.sender];
88          // if there are staked amount, fully harvest current reward
89          if (user.amount > 0) {
90              claim();
91          }
92
93          if (totalTokens > 0) {
94              accRewardsPerShare = accRewardsPerShare + (rewardsPerBlock *
    getMultiplier()) / totalTokens;
95          } else {
96              calculateRewardsDistribution(); // Means first time any user
    deposits, so start distributing
97          }
98
99          lastRewardCalculatedBlock = block.number;
100
101         totalTokens = totalTokens + amount_; // sum of total staked amount
102         user.amount = user.amount + amount_; // cache staked amount count for
    this wallet
103         uint256 lastRewardDebt = user.rewardDebt;
104         user.rewardDebt = (accRewardsPerShare * user.amount) / 1e12; // cache
    current total reward per token
105         allUsersRewardDebt = allUsersRewardDebt + user.rewardDebt -
    lastRewardDebt;
106         emit Deposit(msg.sender, amount_);
107         // Transfer BLS amount from the user to this contract
108         blsToken.safeTransferFrom(address(msg.sender), address(this), amount_);
109     }
```

Then, it is recommended to set the user's reward debt to 0 on withdrawal.

**BlocksStaking.sol**

```
113   function withdraw() public {
114         UserInfo storage user = userInfo[msg.sender];
115         require(user.amount > 0, "No amount deposited for withdrawal.");
116         // Claim any available rewards
117         claim();
```

```
118
119         // Update rewards per share because total tokens change
120         accRewardsPerShare = accRewardsPerShare + (rewardsPerBlock *
    getMultiplier()) / totalTokens;
121         lastRewardCalculatedBlock = block.number;
122
123         uint256 amount = user.amount;
124         totalTokens = totalTokens - amount;
125         user.amount = 0;
126         allUsersRewardDebt = allUsersRewardDebt - user.rewardDebt;
127         user.rewardDebt = 0;
128
129         // Transfer BLS amount from this contract to the user
130         uint256 amountWithdrawn = safeBlsTransfer(address(msg.sender), amount);
131         emit Withdraw(msg.sender, amountWithdrawn);
132     }
```

Finally, we recommend changing the calculation of reward debt and updating the total reward debt on reward claiming.

**BlocksStaking.sol**

```
135 function claim() public {
136         uint256 reward = pendingRewards(msg.sender);
137
138         if (reward <= 0) return; // skip if no rewards
139
140         UserInfo storage user = userInfo[msg.sender];
141         uint256 lastRewardDebt = user.rewardDebt;
142         user.rewardDebt = (accRewardsPerShare * user.amount) / 1e12; // reset:
    cache current total reward per token
143         allUsersRewardDebt = allUsersRewardDebt + user.rewardDebt -
    lastRewardDebt;
144         user.takeoverReward = 0; // reset takeover reward
145
146         // transfer reward in BNBs to the user
147         (bool success, ) = msg.sender.call{value: reward}("");
148         require(success, "Transfer failed.");
149         emit Claim(msg.sender, reward);
150     }
```

Please note that the remediations for other issues are not yet applied to the example above.

## 5.4. Incorrect Reward Calculation from Total Rewards Rate

| | |
|---|---|
| **ID** | IDX-004 |
| **Target** | BlocksRewardsManager |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-840: Business Logic Errors |
| **Risk** | **Severity: High**<br><br>**Impact: Medium**<br>The miscalculation can cause some $BLS rewards to be stuck in the contract and unclaimable by the users.<br><br>**Likelihood: High**<br>The miscalculation can occur whenever there is more than one space. |
| **Status** | **Resolved**<br>The 1000Blocks team has resolved this issue as suggested in commit `8fd0fc2e31dde8c2a214a82ecd9eb10c442415a6` by implementing `blsPerBlock` state variable that is updated using the reward and number of area from each space. |

### 5.4.1. Description

In the `BlocksRewardsManager` contract, the `allBlsPerBlockAreaPerBlock` and `allAllocationBlocks` variables are multiplied to calculate the total $BPS reward to be distributed per block in multiple locations in the contract, for example, in line 284:

**BlocksRewardsManager.sol**

```
276 function depositBlsRewardsForDistribution(uint256 amount_) external onlyOwner {
277     blsToken.transferFrom(address(msg.sender), address(this), amount_);
278
279     blsRewardsAcc = blsRewardsAcc + (block.number -
    blsRewardsAccLastUpdatedBlock) * allAllocationBlocks *
    allBlsPerBlockAreaPerBlock;
280     blsRewardsAccLastUpdatedBlock = block.number;
281     uint256 blsBalance = blsToken.balanceOf(address(this));
282     if (blsBalance > blsRewardsAcc && allAllocationBlocks > 0) {
283         uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
    blsRewardsAcc) /
284             (allBlsPerBlockAreaPerBlock * allAllocationBlocks);
285         blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
286     }
287 }
```

The `allBlsPerBlockAreaPerBlock` comes from the sum of `blsPerBlockAreaPerBlock` from all spaces.

**BlocksRewardsManager.sol**

```
69  function addSpace(address spaceContract_, uint256 blsPerBlockAreaPerBlock_)
    external onlyOwner {
70      spacesByAddress[spaceContract_] = true;
71      uint256 spaceId = spaceInfo.length;
72      SpaceInfo storage newSpace = spaceInfo.push();
73      newSpace.contractAddress = spaceContract_;
74      newSpace.spaceId = spaceId;
75      newSpace.blsPerBlockAreaPerBlock = blsPerBlockAreaPerBlock_;
76      allBlsPerBlockAreaPerBlock = allBlsPerBlockAreaPerBlock +
    blsPerBlockAreaPerBlock_;
77      emit SpaceAdded(spaceId, spaceContract_, msg.sender);
78  }
```

Furthermore, whenever new block areas are bought, the `blocksAreaBoughtOnSpace` will be executed. Then, this function will update the `allAllocationBlocks` variable to count the total number of block areas owned by all users in line 155.

**BlocksRewardsManager.sol**

```
146 uint256 numberOfBlocksAdded = numberOfBlocksBought - numberOfBlocksToRemove;
147 // If amount of blocks on space changed, we need to update space and global
    state
148 if (numberOfBlocksAdded > 0) {
149     blsRewardsAcc =
150         blsRewardsAcc +
151         (block.number - blsRewardsAccLastUpdatedBlock) *
152         allAllocationBlocks *
153         allBlsPerBlockAreaPerBlock;
154     blsRewardsAccLastUpdatedBlock = block.number;
155     allAllocationBlocks = allAllocationBlocks + numberOfBlocksAdded;
156     space.amountOfBlocksBought = space.amountOfBlocksBought +
    numberOfBlocksAdded;
```

Therefore, calculating the total $BLS reward per block using `allBlsPerBlockAreaPerBlock` and `allAllocationBlocks` will cause the calculation to be incorrect because the reward rates for different spaces can be different, and the rates are not meant to be cumulative.

To demonstrate the impact, please see the following example:

Assuming the states to be as follows:

| Space | blsPerBlockAreaPerBlock | amountOfBlocksBought |
|---|---|---|
| 0 | 1 | 1000 |
| 1 | 2 | 1 |

The `allBlsPerBlockAreaPerBlock` variable is equal to 1 + 2 = 3, and the `allAllocationBlocks` variable is equal to 1000 + 1 = 1001.

The total $BLS per block is calculated as follows:

```
Total $BLS per block = allBlsPerBlockAreaPerBlock * allAllocationBlocks = 3 * 1001 =
3003 $BLS per block
```

However, the actual rate should be calculated separately in each space as follows:

```
Space 0 $BLS per block = Space 0 blsPerBlockAreaPerBlock * Space 0
amountOfBlocksBought = 1 * 1000 = 1000
Space 1 $BLS per block = Space 1 blsPerBlockAreaPerBlock * Space 1
amountOfBlocksBought = 2 * 1 = 2

Total $BLS per block = Space 0 $BLS per block + Space 1 $BLS per block = 1000 + 2 =
1002 $BLS per block
```

With the incorrect calculation, the `blsRewardsFinishedBlock` variable will be calculated to be less than what it should be, for example, in line 283-285.

**BlocksRewardsManager.sol**

```
276 function depositBlsRewardsForDistribution(uint256 amount_) external onlyOwner {
277     blsToken.transferFrom(address(msg.sender), address(this), amount_);
278
279     blsRewardsAcc = blsRewardsAcc + (block.number -
    blsRewardsAccLastUpdatedBlock) * allAllocationBlocks *
    allBlsPerBlockAreaPerBlock;
280     blsRewardsAccLastUpdatedBlock = block.number;
281     uint256 blsBalance = blsToken.balanceOf(address(this));
282     if (blsBalance > blsRewardsAcc && allAllocationBlocks > 0) {
283         uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
    blsRewardsAcc) /
284             (allBlsPerBlockAreaPerBlock * allAllocationBlocks);
285         blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
286     }
287 }
```

Since the miscalculated finish block is used in multiplier calculation, some rewards will be stuck in the contract, unclaimable by the users.

**BlocksRewardsManager.sol**

```
 99 function getMultiplier(uint256 usersLastRewardsCalculatedBlock) internal view
    returns (uint256) {
100     if (block.number > blsRewardsFinishedBlock) {
101         return blsRewardsFinishedBlock - usersLastRewardsCalculatedBlock;
102     } else {
103         return block.number - usersLastRewardsCalculatedBlock;
104     }
105 }
```

### 5.4.2. Remediation

Inspex suggests fixing the calculation by removing all usages of `allBlsPerBlockAreaPerBlock` and `allAllocationBlocks` variables and calculating the rate of each space separately. For example:

First, we suggest creating a new variable `blsPerBlock` to store the total rate.

**BlocksRewardsManager.sol**

```
39 // Variables that support calculation of proper bls rewards distributions
40 uint256 public blsPerBlock;
41 uint256 public blsRewardsFinishedBlock;
42 uint256 public blsRewardsAcc; // bls rewards accumulated
43 uint256 public blsRewardsAccLastUpdatedBlock;
44 uint256 public blsRewardsClaimed;
```

Next, in the `updateBlsPerBlockAreaPerBlock()` function, it is recommended to update `blsPerBlock` when the total rate is changed.

**BlocksRewardsManager.sol**

```
80 function updateBlsPerBlockAreaPerBlock(uint256 spaceId_, uint256 newAmount_)
   external onlyOwner {
81     SpaceInfo storage space = spaceInfo[spaceId_];
82     require(space.contractAddress != address(0), "SpaceInfo does not exist");
83     uint256 oldSpaceBlsPerBlock = space.blsPerBlockAreaPerBlock *
   space.amountOfBlocksBought;
84     uint256 newSpaceBlsPerBlock = newAmount_ * space.amountOfBlocksBought;
85     blsPerBlock = blsPerBlock + newSpaceBlsPerBlock - oldSpaceBlsPerBlock; //
   Remove old amount and add new amount
86     space.blsPerBlockAreaPerBlock = newAmount_;
87 }
```

Then, in the `blocksAreaBoughtOnSpace()` function, we recommend updating `blsPerBlock` when the total rate is changed.

**BlocksRewardsManager.sol**

```
148 if (numberOfBlocksAdded > 0) {
149     blsRewardsAcc =
150         blsRewardsAcc +
151         (block.number - blsRewardsAccLastUpdatedBlock) * blsPerBlock;
152     blsRewardsAccLastUpdatedBlock = block.number;
153     uint256 oldSpaceBlsPerBlock = space.blsPerBlockAreaPerBlock *
    space.amountOfBlocksBought;
154     uint256 newSpaceBlsPerBlock = space.blsPerBlockAreaPerBlock *
    space.amountOfBlocksBought + numberOfBlocksAdded;
155     blsPerBlock = blsPerBlock + newSpaceBlsPerBlock - oldSpaceBlsPerBlock;
156     space.amountOfBlocksBought = space.amountOfBlocksBought +
    numberOfBlocksAdded;
157     // Recalculate what is last block eligible for BLS rewards
158     uint256 blsBalance = blsToken.balanceOf(address(this));
159     // If this is true, we are still in state of distribution of rewards
160     if (blsBalance > blsRewardsAcc) {
161         uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
    blsRewardsAcc) / blsPerBlock;
162         blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
163     }
164 }
```

Finally, in the `depositBlsRewardsForDistribution()` function, we suggest using the `blsPerBlock` variable instead.

```
276 function depositBlsRewardsForDistribution(uint256 amount_) external onlyOwner {
277     blsToken.transferFrom(address(msg.sender), address(this), amount_);
278
279     blsRewardsAcc = blsRewardsAcc + (block.number -
    blsRewardsAccLastUpdatedBlock) * blsPerBlock;
280     blsRewardsAccLastUpdatedBlock = block.number;
281     uint256 blsBalance = blsToken.balanceOf(address(this));
282     if (blsBalance > blsRewardsAcc && blsPerBlock > 0) {
283         uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
    blsRewardsAcc) / blsPerBlock;
284         blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
285     }
286 }
```

Please note that the remediations for other issues are not yet applied to the example above.

## 5.5. Token Stealing via BlocksRewardsManager Address Setting

| | |
|---|---|
| **ID** | IDX-005 |
| **Target** | BlocksSpace |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-284: Improper Access Control |
| **Risk** | **Severity:** Medium<br><br>**Impact:** Medium<br>The $BNB that is used to purchase block areas on the contract can be stolen by the owner.<br><br>**Likelihood:** Medium<br>The owner can perform this action freely. However, it will be successful only if the block areas are bought. |
| **Status** | Resolved<br>The 1000Blocks team has resolved this issue as suggested in commit `5565c1bd94ff2378439e82feec6274cb4c13e01c` by removing the `updateRewardsPoolContract()` function. |

### 5.5.1. Description

In the `BlocksSpace` contract, the `purchaseBlocksArea()` function is used for allowing the users to purchase the block areas on the space. The function will send $BNB to an external contract (`rewardPool`) at the last step to perform rewards distribution calculation.

**BlocksSpace.sol**

```
68  function purchaseBlocksArea(
69      uint256 startBlockId_,
70      uint256 endBlockId_,
71      string calldata imghash_
72  ) external payable {
73      BlockAreaLocation memory areaLoc = BlockAreaLocation(
74          startBlockId_ / 100,
75          startBlockId_ % 100,
76          endBlockId_ / 100,
77          endBlockId_ % 100
78      );
79
80      // 1. Checks
81      uint256 paymentReceived = msg.value;
82      require(paymentReceived > 0, "Money expected...");
83      require(
84          block.timestamp >= users[msg.sender].lastPurchase +
```

```
    minTimeBetweenPurchases,
85          "You must wait between buys"
86      );
87      require(isBlocksAreaValid(areaLoc), "BlocksArea invalid");
88      require(bytes(imghash_).length != 0, "Image hash cannot be empty");
89
90      (uint256 currentPriceOfBlocksArea, uint256 numberOfBlocks) =
    calculatePriceAndSize(areaLoc);
91
92      // Price increase per block needs to be at least minimal
93      uint256 priceIncreasePerBlock_ = (paymentReceived -
    currentPriceOfBlocksArea) / numberOfBlocks;
94      require(priceIncreasePerBlock_ > 0, "Price increase too small");
95
96      // 2. Storage operations
97      (address[] memory previousBlockOwners, uint256[] memory
    previousOwnersPrices) = calculateBlocksOwnershipChanges(
98          areaLoc,
99          priceIncreasePerBlock_,
100         numberOfBlocks
101     );
102     updateUserState(msg.sender, startBlockId_, endBlockId_, imghash_);
103
104     // 3. Transactions
105     // Send fresh info to RewardsPool contract, so buyer gets some sweet
    rewards
106     rewardsPool.blocksAreaBoughtOnSpace{value: paymentReceived}(
107         spaceId,
108         msg.sender,
109         previousBlockOwners,
110         previousOwnersPrices
111     );
112
113     // 4. Emit purchase event
114     emit BlocksAreaPurchased(msg.sender, startBlockId_ * 10000000 +
115 endBlockId_, paymentReceived);
    }
```

However, the owner can set `rewardPool` to be any address in the `updateRewardsPoolContract()` function.

**BlocksSpace.sol**

```
211 function updateRewardsPoolContract(address add_) external onlyOwner {
212     rewardsPool = BlocksRewardsManager(add_);
213 }
```

This allows the newly set contract to maliciously receive $BNB from the users.

### 5.5.2. Remediation

Inspex suggests removing the `updateRewardsPoolContract()` function from the `BlocksSpace` contract and setting the `rewardPool` address from the constructor.

## 5.6. Token Stealing via BlocksStaking Address Setting

| | |
|---|---|
| **ID** | IDX-006 |
| **Target** | BlocksRewardsManager |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-284: Improper Access Control |
| **Risk** | **Severity:** Medium<br><br>**Impact:** Medium<br>The $BNB that is used to purchase block areas from `BlocksSpace` contract can be all stolen after forwarding to this contract.<br><br>**Likelihood:** Medium<br>The owner can perform this action freely. However, it will be successful only if the block areas are bought. |
| **Status** | Resolved *<br>**[Sep 17, 2021]**<br>The 1000Blocks team has mitigated this issue by implementing a 24-hours `Timelock` over the `BlocksRewardsManager` contract.<br><br>The `BlocksRewardsManager` contract has been deployed at the following address: `0x198012cDfBDfb2EF58674545f7D147d928Ff5ecC`.<br><br>It is owned by a `Timelock` contract deployed at the following address: `0xfdf6b2D7E24912f5B426741C33737B797d5ef420`<br><br>**[Aug 25, 2021]**<br>The 1000Blocks team has confirmed that they will mitigate this issue by implementing the timelock mechanism when deploying the smart contracts to mainnet. The users will be able to monitor the timelock for the execution of `updateBlocksStakingContract()` function and act accordingly if it is being misused.<br><br>At the time of the reassessment, the contracts are not deployed yet, so the use of timelock is not confirmed. For the platform users, please verify that the timelock is properly deployed before using this platform.<br><br>Please note that the `updateBlocksStatingContract()` function has been renamed to `updateBlocksStakingContract()`. |

### 5.6.1. Description

In the `BlocksRewardsMananger` contract, the `blocksAreaBoughtOnSpace()` function is used to calculate rewards for both the buyer and the previous owners of the bought block areas, including fees. After the calculation is successful, the `blocksAreaBoughtOnSpace()` function will call an external contract (`blocksStaking`) to distribute rewards to the users who stake $BLS and the previous owner.

BlocksStaking.sol

```
107 function blocksAreaBoughtOnSpace(
108     uint256 spaceId_,
109     address buyer_,
110     address[] calldata previousBlockOwners_,
111     uint256[] calldata previousOwnersPrices_
112 ) public payable onlySpace {
113     SpaceInfo storage space = spaceInfo[spaceId_];
114     UserInfo storage user = userInfo[spaceId_][buyer_];
115     uint256 blsPerBlockAreaPerBlock = space.blsPerBlockAreaPerBlock;
116 
117     // If user already had some block.areas then calculate all rewards pending
118     if (user.lastRewardCalculatedBlock > 0) {
119         uint256 multiplier = getMultiplier(user.lastRewardCalculatedBlock);
120         uint256 blsRewards = multiplier * blsPerBlockAreaPerBlock;
121         user.pendingRewards = user.pendingRewards + user.amount * blsRewards;
122     }
123     uint256 numberOfBlocksBought = previousBlockOwners_.length;
124     // Set user data
125     user.amount = user.amount + numberOfBlocksBought;
126     user.lastRewardCalculatedBlock = block.number;
127 
128     //remove blocks from previous owners that this guy took over. Max 42 loops
129     uint256 allPreviousOwnersPaid;
130     uint256 numberOfBlocksToRemove;
131     for (uint256 i = 0; i < numberOfBlocksBought; ++i) {
132         // If previous owners of block are non zero address, means we need to
    take block from them
133         if (previousBlockOwners_[i] != address(0)) {
134             allPreviousOwnersPaid = allPreviousOwnersPaid +
    previousOwnersPrices_[i];
135             // Calculate previous users pending BLS rewards
136             UserInfo storage prevUser =
    userInfo[spaceId_][previousBlockOwners_[i]];
137             uint256 multiplier =
    getMultiplier(prevUser.lastRewardCalculatedBlock);
138             uint256 blsRewards = multiplier * blsPerBlockAreaPerBlock;
139             prevUser.pendingRewards = prevUser.pendingRewards + prevUser.amount
    * blsRewards;
140             prevUser.lastRewardCalculatedBlock = block.number;
141             // Remove his ownership of block
142             --prevUser.amount;
143             ++numberOfBlocksToRemove;
144         }
145     }
146     uint256 numberOfBlocksAdded = numberOfBlocksBought -
    numberOfBlocksToRemove;
```

```
147     // If amount of blocks on space changed, we need to update space and global
    state
148     if (numberOfBlocksAdded > 0) {
149         blsRewardsAcc =
150             blsRewardsAcc +
151             (block.number - blsRewardsAccLastUpdatedBlock) *
152             allAllocationBlocks *
153             allBlsPerBlockAreaPerBlock;
154         blsRewardsAccLastUpdatedBlock = block.number;
155         allAllocationBlocks = allAllocationBlocks + numberOfBlocksAdded;
156         space.amountOfBlocksBought = space.amountOfBlocksBought +
    numberOfBlocksAdded;
157         // Recalculate what is last block eligible for BLS rewards
158         uint256 blsBalance = blsToken.balanceOf(address(this));
159         // If this is true, we are still in state of distribution of rewards
160         if (blsBalance > blsRewardsAcc) {
161             uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
    blsRewardsAcc) /
162                 (allBlsPerBlockAreaPerBlock * allAllocationBlocks);
163             blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
164         }
165     }
166
167     // Calculate and subtract fees in first part
168     // In second part, calculate how much rewards are being rewarded to
    previous block owners
169     (uint256 rewardToForward, uint256[] memory prevOwnersRewards) =
    calculateAndDistributeFees(
170         msg.value,
171         previousOwnersPrices_,
172         allPreviousOwnersPaid
173     );
174
175     // Send to distribution part
176     blocksStaking.distributeRewards{value:
    rewardToForward}(previousBlockOwners_, prevOwnersRewards);
177 }
```

However, the owner can set **blocksStaking** to be any address with the **updateBlocksStatingContract()** function.

**BlocksRewardsManager.sol**

```
268 function updateBlocksStatingContract(address address_) external onlyOwner {
269     blocksStaking = BlocksStaking(address_);
270 }
```

This allows the newly set contract to maliciously receive $BNB from the users.

### 5.6.2. Remediation

Inspex suggests removing the `updateBlocksStatingContract()` function from the `BlocksRewardsManager` contract and setting the `blocksStaking` address once from the constructor function.

## 5.7. Centralized Control of State Variable

| | |
|---|---|
| **ID** | IDX-007 |
| **Target** | BlocksRewardsManager<br>BlocksSpace<br>BlocksStaking |
| **Category** | General Smart Contract Vulnerability |
| **CWE** | CWE-710: Improper Adherence to Coding Standard |
| **Risk** | **Severity:** Medium<br><br>**Impact:** Medium<br>The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users.<br><br>**Likelihood:** Medium<br>There is nothing to restrict the changes from being done by the owner; however, the changes are limited by fixed values in the smart contracts. |
| **Status** | Resolved<br>**[Sep 17, 2021]**<br>The 1000Blocks team has resolved this issue by implementing a 24-hours `Timelock` over the following contracts:<br><br>- `BlocksSpace`<br>Contract Address: `0xB2C159d81AFE012636A322F584D743919d58652c`<br>Owner Address (Timelock): `0xfdf6b2D7E24912f5B426741C33737B797d5ef420`<br>- `BlocksRewardsManager`<br>Contract Address: `0x198012cDfBDfb2EF58674545f7D147d928Ff5ecC`<br>Owner Address (Timelock): `0xfdf6b2D7E24912f5B426741C33737B797d5ef420`<br>- `BlocksStaking`<br>Contract Address: `0x353aAb6Ad697c970dDc1378F190BbCc4cEB86C1e`<br>Owner Address (Timelock): `0xfdf6b2D7E24912f5B426741C33737B797d5ef420`<br><br>**[Aug 25, 2021]**<br>The 1000Blocks team has confirmed that they will implement the timelock mechanism. The users will be able to monitor the timelock for the execution of privileged functions and act accordingly.<br><br>At the time of the reassessment, the contracts are not deployed yet, so the use of timelock is not confirmed. For the platform users, please verify that the timelock is properly deployed before using this platform. |

### 5.7.1. Description

Critical state variables can be updated any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, as the contract is not yet deployed, there is potentially no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

| File | Contract | Function | Modifier |
|---|---|---|---|
| BlocksSpace.sol (L:211) | BlocksSpace | updateRewardsPoolContract() | onlyOwner |
| BlocksSpace.sol (L:215) | BlocksSpace | updateMinTimeBetweenPurchases() | onlyOwner |
| BlocksStaking.sol (L:52) | BlocksStaking | setRewardDistributionPeriod() | onlyOwner |
| BlocksRewardsManager.sol (L:253) | BlocksRewardsManager | setTreasuryFee() | onlyOwner |
| BlocksRewardsManager.sol (L:258) | BlocksRewardsManager | setLiquidityFee() | onlyOwner |
| BlocksRewardsManager.sol (L:263) | BlocksRewardsManager | setPreviousOwnerFee() | onlyOwner |
| BlocksRewardsManager.sol (L:268) | BlocksRewardsManager | updateBlocksStatingContract() | onlyOwner |
| BlocksRewardsManager.sol (L:272) | BlocksRewardsManager | updateTreasuryWallet() | onlyOwner |

### 5.7.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests limiting the use of these functions via the following options:

- Implementing a community-run governance to control the use of these functions
- Using a `Timelock` contract to delay the changes for a sufficient amount of time

## 5.8. Incorrect Condition

| | |
|---|---|
| **ID** | IDX-008 |
| **Target** | BlocksRewardsManager |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-840: Business Logic Errors |
| **Risk** | **Severity:** Medium<br><br>**Impact:** Medium<br>The `blsRewardsFinishedBlock` will not be updated properly, causing the $BLS reward to be incorrectly distributed.<br><br>**Likelihood:** Medium<br>This issue will cause effect when the total reward rate is changed or more $BLS are deposited to the contract for distribution, and more than half of the $BLS reward is distributed. |
| **Status** | Resolved<br>The 1000Blocks team has resolved this issue as suggested in commit `215da1e2c09c4bb322d70375671b42a49ac139a6` |

### 5.8.1. Description

In the `BlocksRewardManager` contract, the `blsBalance > blsRewardsAcc` was used as a condition to check whether there is enough $BLS to be distributed or not.

For example, it is used in line 160 of the `blocksAreaBoughtOnSpace()` function, and in line 282 of the `depositBlsRewardsForDistribution()` function.

**BlocksRewardsManager.sol**

```
159 // If this is true, we are still in state of distribution of rewards
160 if (blsBalance > blsRewardsAcc) {
161     uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
    blsRewardsAcc) /
162         (allBlsPerBlockAreaPerBlock * allAllocationBlocks);
163     blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
164 }
```

**BlocksRewardsManager.sol**

```
276 function depositBlsRewardsForDistribution(uint256 amount_) external onlyOwner {
277     blsToken.transferFrom(address(msg.sender), address(this), amount_);
278
279     blsRewardsAcc = blsRewardsAcc + (block.number -
```

```
    blsRewardsAccLastUpdatedBlock) * allAllocationBlocks *
    allBlsPerBlockAreaPerBlock;
280     blsRewardsAccLastUpdatedBlock = block.number;
281     uint256 blsBalance = blsToken.balanceOf(address(this));
282     if (blsBalance > blsRewardsAcc && allAllocationBlocks > 0) {
283         uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
284     blsRewardsAcc) /
                (allBlsPerBlockAreaPerBlock * allAllocationBlocks);
285         blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
286     }
287 }
```

However, this condition does not include the amount claimed by the user, causing the `blsRewardsFinishedBlock` variable to not be updated even when there is enough $BLS to be distributed.

Please consider the following example case, assuming that:

- Original `blsBalance` = 250
- `blsRewardsAcc` = 200
- `blsRewardsClaimed` = 150
- Remaining `blsBalance` = 100

This means that 50 $BLS is still available for distribution; however, since `blsBalance` > `blsRewardsAcc` condition is not fulfilled, `blsRewardsFinishedBlock` won't be updated.

The `blsRewardsFinishedBlock` state variable is used in the calculation of the multiplier. If it is incorrect, the $BLS reward will be incorrectly distributed.

**BlocksRewardsManager.sol**

```
 99 function getMultiplier(uint256 usersLastRewardsCalculatedBlock) internal view
    returns (uint256) {
100     if (block.number > blsRewardsFinishedBlock) {
101         return blsRewardsFinishedBlock - usersLastRewardsCalculatedBlock;
102     } else {
103         return block.number - usersLastRewardsCalculatedBlock;
104     }
105 }
```

### 5.8.2. Remediation

Inspex suggests including the claimed amount in the calculation as shown in the following examples:

**BlocksRewardsManager.sol**

```
159 // If this is true, we are still in state of distribution of rewards
160 if (blsBalance + blsRewardsClaimed > blsRewardsAcc) {
161     uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
    blsRewardsAcc) /
162         (allBlsPerBlockAreaPerBlock * allAllocationBlocks);
163     blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
164 }
```

**BlocksRewardsManager.sol**

```
276 function depositBlsRewardsForDistribution(uint256 amount_) external onlyOwner {
277     blsToken.transferFrom(address(msg.sender), address(this), amount_);
278
279     blsRewardsAcc = blsRewardsAcc + (block.number -
    blsRewardsAccLastUpdatedBlock) * allAllocationBlocks *
    allBlsPerBlockAreaPerBlock;
280     blsRewardsAccLastUpdatedBlock = block.number;
281     uint256 blsBalance = blsToken.balanceOf(address(this));
282     if (blsBalance + blsRewardsClaimed > blsRewardsAcc && allAllocationBlocks >
    0) {
283         uint256 blocksTillBlsRunOut = (blsBalance + blsRewardsClaimed -
284 blsRewardsAcc) /
                (allBlsPerBlockAreaPerBlock * allAllocationBlocks);
285         blsRewardsFinishedBlock = block.number + blocksTillBlsRunOut;
286     }
287 }
```

## 5.9. Incorrect Reward Calculation from blsPerBlockAreaPerBlock

| | |
|---|---|
| **ID** | IDX-009 |
| **Target** | BlocksRewardsManager |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-840: Business Logic Errors |
| **Risk** | **Severity:** Low<br><br>**Impact:** Medium<br>The $BLS rewards will be incorrectly calculated on the changing of `blsPerBlockAreaPerBlock`.<br><br>**Likelihood:** Low<br>It is unlikely that the `blsPerBlockAreaPerBlock` will be changed. |
| **Status** | Resolved<br>The 1000Blocks team resolved this issue by redesigning the reward calculation mechanism in the following commits:<br>- `8fd0fc2e31dde8c2a214a82ecd9eb10c442415a6`<br>- `a2e71a0d062a2e8f314933501329eaf60259d7f9` |

### 5.9.1. Description

In the `BlocksRewardsManager` contract, the `blsPerBlockAreaPerBlock` state variable of each space determines the amount of $BLS rewards to be distributed.

**BlocksRewardsManager.sol**

```
219 function claim(uint256 spaceId_) public {
220     UserInfo storage user = userInfo[spaceId_][msg.sender];
221     uint256 amount = user.amount;
222     uint256 lastRewardCalculatedBlock = user.lastRewardCalculatedBlock;
223     if (amount > 0 && lastRewardCalculatedBlock < block.number) {
224         user.pendingRewards =
225             user.pendingRewards +
226             amount *
227             getMultiplier(lastRewardCalculatedBlock) *
228             spaceInfo[spaceId_].blsPerBlockAreaPerBlock;
229         user.lastRewardCalculatedBlock = block.number;
230     }
231     uint256 toClaimAmount = user.pendingRewards;
232     if (toClaimAmount > 0) {
233         uint256 claimedAmount = safeBlsTransfer(msg.sender, toClaimAmount);
234         emit Claim(msg.sender, claimedAmount);
235         // This is also kinda check, since if user claims more than eligible,
```

```
        this will revert
236             user.pendingRewards = toClaimAmount - claimedAmount;
237             blsRewardsClaimed = blsRewardsClaimed + claimedAmount; // Globally
    claimed rewards, for proper end distribution calc
238         }
239     }
```

The `blsPerBlockAreaPerBlock` variable can be updated using the `updateBlsPerBlockAreaPerBlock()` function.

**BlocksRewardsManager.sol**

```
80  function updateBlsPerBlockAreaPerBlock(uint256 spaceId_, uint256 newAmount_)
    external onlyOwner {
81      SpaceInfo storage space = spaceInfo[spaceId_];
82      require(space.contractAddress != address(0), "SpaceInfo does not exist");
83      allBlsPerBlockAreaPerBlock = allBlsPerBlockAreaPerBlock -
    space.blsPerBlockAreaPerBlock + newAmount_; // Remove old amount and Add new
    amount
84      space.blsPerBlockAreaPerBlock = newAmount_;
85  }
```

However, if it is updated, the rewards of the users will be incorrectly calculated.

To demonstrate the impact, assuming that:

- At block 0, the `blsPerBlockAreaPerBlock` is 10, and a user is holding 10 blocks of space.
- At block 10, the contract owner sets `blsPerBlockAreaPerBlock` to 5.
- At block 20, the user claims the $BLS rewards.

Using the following formula:

```
amount * getMultiplier(lastRewardCalculatedBlock) * blsPerBlockAreaPerBlock
```

The user will get:

- **Block 0 -> Block 20** $BLS = 10 * (20-0) * 5 = 1000 $BLS

However, the actual rewards should be calculated with the rate of each period as follows:

- **Block 0 -> Block 10** $BLS = 10 * (10-0) * 10 = 1000 $BLS
- **Block 10 -> Block 20** $BLS = 10 * (20-10) * 5 = 500 $BLS
- **Block 0 -> Block 20** $BLS = 1000 + 500 = 1500 $BLS

### 5.9.2. Remediation

Inspex suggests removing the `updateBlsPerBlockAreaPerBlock()` function to prevent the rewards from being miscalculated.

However, if modifications are needed, it is recommended to redesign how the rewards are calculated for the users. One of the suggested solutions is to implement the state variables to keep track the claimed amount of users and the accumulated rewards per share of each space, for example:

First, we suggests adding the following state variables to the `BlocksRewardsManagers` contract:

- `rewardDebt`: This state variable is used for calculating the reward amount entitled to the user, updating this state variable whenever the user deposits, withdraws, or claims $BLS.
- `accBlsPerShare`: This state variable is used for accumulating the $BLS amount per share of each user in the space, updating this whenever the amount of the reward is calculated.
- `lastRewardBlock`: This state variable is used for tracking the latest block number that the $BLS distribution occurs.

**BlocksRewardsManager.sol**

```
 9  contract BlocksRewardsManager is Ownable {
10      // Info of each user.
11      struct UserInfo {
12          uint256 amount; // How many blocks user owns currently.
13          uint256 rewardDebt;
14      }
15
16      // Info of each blocks.space
17      struct SpaceInfo {
18          uint256 spaceId;
19          uint256 amountOfBlocksBought; // Number of all blocks bought on this
    space
20          address contractAddress; // Address of space contract.
21          uint256 blsPerBlockAreaPerBlock; // Start with 830000000000000 wei
    (approx 24 BLS/block.area/day)
22          uint256 accBlsPerShare;
23          uint256 lastRewardBlock;
24      }
```

Finally, it is recommended to apply `rewardDebt` and `accBlsPerShare` state variables in all reward calculation formulas to ensure the correct distributed rewards for the users.

## 5.10. Insufficient Logging for Privileged Functions

| | |
|---|---|
| **ID** | IDX-010 |
| **Target** | BlocksSpace<br>BlocksRewardsManager<br>BlocksStaking |
| **Category** | Advanced Smart Contract Vulnerability |
| **CWE** | CWE-778: Insufficient Logging |
| **Risk** | **Severity:** Very Low<br><br>**Impact:** Low<br>Privileged functions' executions cannot be monitored easily by the users.<br><br>**Likelihood:** Low<br>It is not likely that the execution of the privileged functions will be a malicious action. |
| **Status** | Resolved<br>The 1000Blocks team has resolved this issue as suggested in the following commits:<br><br>- `5565c1bd94ff2378439e82feec6274cb4c13e01c`<br>- `d8b6f244f8d98e3f9952caeb66cecba362cd6b55`<br>- `a2e71a0d062a2e8f314933501329eaf60259d7f9`<br><br>The `updateRewardsPoolContract()` function has been removed from `BlocksSpace` contract. |

### 5.10.1. Description

Privileged functions that are executable by the controlling parties are not logged properly by emitting events. Without events, it is not easy for the public to monitor the execution of those privileged functions, allowing the controlling parties to perform actions that cause big impacts to the platform.

For example, the owner can set the time delay by executing `updateMinTimeBetweenPurchases()` function in the `BlocksSpace` contract, and no events are emitted, although the updated value would affect the users when they purchase new block areas.

The privileged functions without sufficient logging are as follows:

| File | Contract | Function |
|---|---|---|
| BlocksSpace.sol (L:211) | BlocksSpace | updateRewardsPoolContract() |
| BlocksSpace.sol (L:215) | BlocksSpace | updateMinTimeBetweenPurchases() |

| | | |
|---|---|---|
| BlocksRewardsManager.sol (L:80) | BlocksRewardsManager | updateBlsPerBlockAreaPerBlock() |
| BlocksRewardsManager.sol (L:253) | BlocksRewardsManager | setTreasuryFee() |
| BlocksRewardsManager.sol (L:258) | BlocksRewardsManager | setLiquidityFee() |
| BlocksRewardsManager.sol (L:263) | BlocksRewardsManager | setPreviousOwnerFee() |
| BlocksRewardsManager.sol (L:268) | BlocksRewardsManager | updateBlocksStatingContract() |
| BlocksRewardsManager.sol (L:272) | BlocksRewardsManager | updateTreasuryWallet() |
| BlocksRewardsManager.sol (L:276) | BlocksRewardsManager | depositBlsRewardsForDistribution() |
| BlocksStaking.sol (L:52) | BlocksStaking | setRewardDistributionPeriod() |

### 5.10.2. Remediation

Inspex suggests emitting events for the execution of privileged functions, for example:

**BlocksSpace.sol**

```
214 event UpdateMinTimeBetweenPurchases(uint256 inSeconds_)
215 function updateMinTimeBetweenPurchases(uint256 inSeconds_) external onlyOwner {
216     minTimeBetweenPurchases = inSeconds_;
217     emit UpdateMinTimeBetweenPurchases(inSeconds_);
218 }
```

## 5.11. Outdated Compiler Version

| | |
|---|---|
| **ID** | IDX-011 |
| **Target** | BLSToken |
| **Category** | General Smart Contract Vulnerability |
| **CWE** | CWE-1104: Use of Unmaintained Third Party Components |
| **Risk** | **Severity:** Info<br>**Impact:** None<br>**Likelihood:** None |
| **Status** | No Security Impact<br>The 1000Blocks team has acknowledged this issue and decided to keep the version as 0.8.5 since the contract is already deployed. |

### 5.11.1. Description

The Solidity compiler versions specified in the smart contracts were outdated. These versions have publicly known inherent bugs that may potentially be used to cause damage to the smart contracts or the users of the smart contracts.

**BLSToken.sol**

```
1 pragma solidity 0.8.5;
2 //SPDX-License-Identifier: MIT
```

### 5.11.2. Remediation

Inspex suggests upgrading the Solidity compiler of the BLSToken contract to the latest stable version.

During the audit activity, the latest stable versions of Solidity compiler in each major is v0.8.7.

## 5.12. Improper Function Visibility

| | |
|---|---|
| **ID** | IDX-012 |
| **Target** | BLSToken<br>BlocksRewardsManager<br>BlocksSpace<br>BlocksStaking |
| **Category** | Smart Contract Best Practice |
| **CWE** | CWE-710: Improper Adherence to Coding Standards |
| **Risk** | **Severity:** Info<br>**Impact:** None<br>**Likelihood:** None |
| **Status** | No Security Impact<br>The 1000Blocks team has resolved this issue as suggested in the following commits:<br>- `c07e953313e13f1f3ba89238bef639a899817470`<br>- `a2e71a0d062a2e8f314933501329eaf60259d7f9`<br>The `BlocksRewardsManager.pendingBlsTokens()` is now called within the contract, so the visibility is not changed. However, as the `BLSToken` contract is already deployed, the `burn()` function is not changed. |

### 5.12.1. Description

Functions with public visibility copy calldata to memory when being executed, while external functions can read directly from calldata. Memory allocation uses more resources (gas) than reading directly from calldata.

For example, the following source code shows that the `burn()` function of the `BLSToken` contract is set to public and it is never called from any internal function.

**BLSToken.sol**

```
14 function burn(uint256 amount) public virtual {
15     _burn(_msgSender(), amount);
16 }
```

The following table contains all functions that have `public` visibility and are never called from any internal function.

| File | Contract | Function |
|---|---|---|
| BLSToken.sol (L:14) | BLSToken | burn() |

| | | |
|---|---|---|
| BlocksRewardsManager.sol (L:87) | BlocksRewardsManager | pendingBlsTokens() |
| BlocksRewardsManager.sol (L:107) | BlocksRewardsManager | blocksAreaBoughtOnSpace() |
| BlocksRewardsManager.sol (L:219) | BlocksRewardsManager | claim() |
| BlocksStaking.sol (L:86) | BlocksStaking | deposit() |
| BlocksStaking.sol (L:113) | BlocksStaking | withdraw() |
| BlocksStaking.sol (L:153) | BlocksStaking | distributeRewards() |

### 5.12.2. Remediation

Inspex suggests changing all functions' visibility to `external` if they are not called from any `internal` function as shown in the following example:

**BLSToken.sol**

```
14 function burn(uint256 amount) external virtual {
15     _burn(_msgSender(), amount);
16 }
```

## 5.13. Inexplicit Solidity Compiler Version

| | |
|---|---|
| **ID** | IDX-013 |
| **Target** | BlocksRewardsManager<br>BlocksSpace<br>BlocksStaking |
| **Category** | Smart Contract Best Practice |
| **CWE** | CWE-1104: Use of Unmaintained Third Party Components |
| **Risk** | **Severity:** Info<br>**Impact:** None<br>**Likelihood:** None |
| **Status** | Resolved<br>The 1000Blocks team has resolved this issue in commit `47d3714345028210542ae7cd57103e0df5fb661c` by explicitly specifying the Solidity version to 0.8.5 to match the already deployed `BLSToken` contract. |

### 5.13.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues, for example:

**BlocksRewardsManager.sol**

```
1 pragma solidity ^0.8.0;
2 //SPDX-License-Identifier: MIT
```

The following table contains all targets which the inexplicit compiler version is declared.

| Contract | Version |
|---|---|
| BlocksRewardsManager | ^0.8.0 |
| BlocksSpace | ^0.8.0 |
| BlocksStaking | ^0.8.0 |

### 5.13.2. Remediation

Inspex suggests fixing the solidity compiler to the latest stable version. At the time of audit, the latest stable version of Solidity compiler in major 0.8 is v0.8.7.

# 6. Appendix

## 6.1. About Inspex


Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

**Follow Us On:**

| | |
|---|---|
| **Website** | https://inspex.co |
| **Twitter** | @InspexCo |
| **Facebook** | https://www.facebook.com/InspexCo |
| **Telegram** | @inspex_announcement |

## 6.2. References

[1] “OWASP Risk Rating Methodology.” [Online]. Available: https://owasp.org/www-community/OWASP_Risk_Rating_Methodology. [Accessed: 08-May-2021]

inspeX

CYBERSECURITY PROFESSIONAL SERVICE