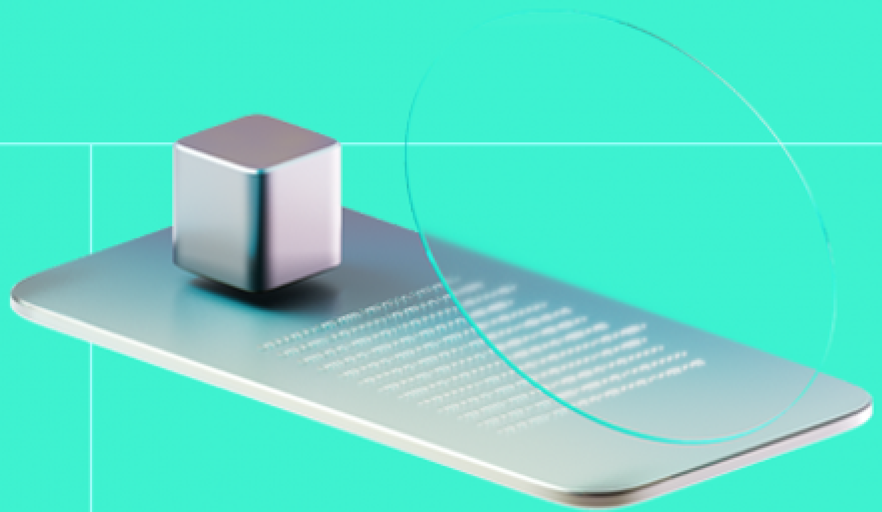
Smart Contract Code Review And Security Analysis Report



Customer: BlockSquare

Date: 20/02/2024

We express our gratitude to the BlockSquare team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Blocksquare is a comprehensive real estate tokenization platform designed to cater to a diverse range of businesses, from those with extensive international real estate portfolios to boutique investment clubs focusing on local investment models.

Platform: EVM

Language: Solidity

Tags: Factory; Upgradable; Proxy; Yield Farming

Timeline: 26/01/2024 - 20/02/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/blocksquare/oceanpoint-contracts/
Commit	d9c5ebf

Audit Summary

10/10	10/10	85.6%	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.5/10

The system users should acknowledge all the risks summed up in the risks section of the report

8	8	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Severity	Count
Critical	1
High	0
Medium	2
Low	4

Vulnerability	Status
F-2024-0629 - Inconsistent Use of Upgradeable Contracts and Incomplete Initialization in MarketplacePool	Fixed
F-2024-0631 - Variable Shadowing in MarketplacePool Contract	Fixed
F-2024-0635 - Lack of Validation for Start Time and Duration in MarketplacePool's Campaign	Fixed
F-2024-0658 - Incorrect Maximum Pledge Calculation in depositInCampaign Function	Fixed
F-2024-0659 - Checks-Effects-Interactions Pattern Violation in _deposit Function	Fixed
F-2024-0661 - Restrictive CP Wallet Collateral Withdrawal Logic in MarketplacePool Contract	Fixed
F-2024-0662 - Owner-Controlled Liquidation of CP Collateral	Fixed
F-2024-0664 - Calculation Discrepancy in MarketplacePool Post-Campaign Distribution	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for BlockSquare
Audited By	Ivan Bondar
Approved By	Ataberk Yavuzer
Website	https://blocksquare.io/
Changelog	30/01/2024 - Preliminary Report
	20/02/2024 - Final Report

Table of Contents

System Overview **6**
Privileged Roles 6
Executive Summary **7**
Documentation Quality 7
Code Quality 7
Test Coverage 7
Security Score 7
Summary 7
Risks **8**
Findings **9**
Vulnerability Details 9
Observation Details 29
Disclaimers 35
Appendix 1. Severity Definitions **36**
Appendix 2. Scope **37**

System Overview

The Blocksquare Marketplace Pool System represents a sophisticated smart contract framework, aimed at streamlining investment and reward distribution processes in real estate projects. This system is uniquely structured to balance the interests of Certified Partners (CPs) and investors, promoting transparent and secure financial interactions.

Files in the scope:

- MarketplacePoolProxyFactory.sol - functions as the core factory for creating individual Marketplace Pool Proxies. It maintains critical configurations such as logic contract addresses, BST staking contracts, and governance wallets. Key functionalities include:
    - Marketplace Pool Creation: Generates new Marketplace Pool Proxies, each linked to a specific Certified Partner (CP).
    - Configuration Management: Allows for updating key components like implementation logic, BST staking contracts, and governance wallets.
- MarketplacePoolProxy.sol - Serves as a Transparent Upgradeable Proxy for each Marketplace Pool. It leverages the ERC1967Proxy pattern.
- MarketplacePool.sol - the primary contract where investment and reward mechanisms are actualized. It is upgradeable, owned, and reentrancy-guarded, incorporating ERC20 functionalities for internal accounting:
    - CP and Investor Interaction: Facilitates CPs to initialize pools with collateral and set lock periods, while allowing investors to contribute during designated campaign periods.
    - Investment Campaign Management: Manages the campaign's start time, duration, and maximum pledges, dynamically adjusting investment caps.
    - Reward Distribution: Implements a mechanism for distributing rewards based on individual stakes and total pool balance.
    - Collateral and Lock Management: Provides functions for CP collateral withdrawal, liquidation, and lock period extensions, governed by ownership privileges.

Privileged roles

- MarketplacePoolProxyFactory.sol:
    - Owner :
        - Manages the creation of new pools, updates implementation logic, and configures key contracts like BST staking and governance wallet.
- MarketplacePool.sol:
    - Owner:
        - Holds the power to allow CP collateral withdrawal, extend lock periods, and liquidate CP collateral under defined conditions.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements have some gaps:
  - Project overview is detailed.
  - Roles and permissions are described.
  - Use cases are described.
  - For each contract all futures are described.
- Technical description is robust:
  - Run instructions are provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.

Test coverage

Code coverage of the project is **85.6%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is partially missed.
- Interactions by several users are not tested thoroughly.

Security score

Upon auditing, the code was found to contain **1** critical, **0** high, **2** medium, and **4** low severity issues. All issues were fixed in the remediation part of this audit, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9.5**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- **Centralized Upgrades**:
  - The factory contract acts as a central point for upgrading the implementation logic of all proxy contracts. By updating the implementation address in the factory contract, all associated proxies will use the new logic.
- **Solidity Version Compatibility and Cross-Chain Deployment**:
  - The project utilizes Solidity version 0.8.20 or higher, which includes the introduction of the `PUSH0` (`0x5f`) opcode. This opcode is currently supported on the Ethereum mainnet but may not be universally supported across other blockchain networks. Consequently, deploying the contract on chains other than the Ethereum mainnet, such as certain Layer 2 (L2) chains or alternative networks, might lead to compatibility issues or execution errors due to the lack of support for the PUSH0 opcode. In scenarios where deployment on various chains is anticipated, selecting an appropriate Ethereum Virtual Machine (EVM) version that is widely supported across these networks is crucial to avoid potential operational disruptions or deployment failures.
- **Uncertainty in Reward Allocation in MarketplacePool**:
  - **Conditional Reward Distribution**: Rewards can only be added to pools that are marked as successful. This introduces an element of uncertainty, as rewards depend on the successful status of the pool.
  - **Dependence on External Action**: Rewards are allocated manually through the `addReward` function and only after the pool ends. This means there's no guaranteed reward during the pool's active period.
  - **Variable Reward Amounts**: The total reward depends on the amount specified in `addReward` call. This can lead to variability in reward amounts, with no predefined or fixed reward structure.

Findings

Vulnerability Details

F-2024-0664 - Calculation Discrepancy in MarketplacePool Post-Campaign Distribution - Critical

Description:

The MarketplacePool contract faces a notable issue with the calculation of user deposits and rewards, primarily after reward distribution. The root of this problem lies in the dual use of the `_tempsBSTBalanceThis` variable, which is employed for tracking both user deposits and calculating rewards. This overlapping usage leads to discrepancies, particularly evident when new deposits are made after rewards was added.

- `_deposit` function: In the `_deposit` function, the calculation for **amountToMint** is based on the ratio of the deposited amount to the total supply, which in turn is influenced by `_tempsBSTBalanceThis`. When new deposits are added post-campaign, they affect the total supply but are rounded down during the minting process. This round-down affects the balance between `totalSupply` and `_tempsBSTBalanceThis`, leading to inconsistencies.

```
function _deposit(uint256 amount) private {
uint256 amountToMint = 0;
if (_msgSender() == _cpWallet) {
//..
} else {
amountToMint = (totalSupply() == 0 || _tempsBSTBalanceThis == 0)
? amount
: (amount * totalSupply()) / _tempsBSTBalanceThis;
_mint(_msgSender(), amountToMint);
_tempsBSTBalanceThis += amount;
}
//..
}
```

- `_withdrawUser` function: The `_withdrawUser` function demonstrates where these discrepancies become problematic. It calculates the amount a user is entitled to withdraw (`amountToSend`) and their rewards, based on their share of the total supply and `_tempsBSTBalanceThis`. For users who deposit after the campaign, the mismatch in `amountToMint` and `_tempsBSTBalanceThis` can result in their withdrawal amount being less than their deposit, due to the rounding down in minting. This situation can lock their funds in the contract.

```
function _withdrawUser() private returns (uint256) {
//..
uint256 userDeposited = _deposited[_msgSender()];
uint256 share = balanceOf(_msgSender());
uint256 amountToSend = (share * _tempsBSTBalanceThis) / totalSupply();
uint256 reward = amountToSend - userDeposited;
//..
}
```

The overlapping use of `_tempsBSTBalanceThis` for both deposit tracking and reward calculation creates a significant issue. It can lead to skewed reward distributions and, more critically, the potential locking of users' deposited tokens

and rewards within the contract. The problem is accentuated in scenarios involving post-campaign deposits, where the new deposits alter the reward calculations for existing users and can prevent new depositors from withdrawing their funds.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Classification

Severity: Critical

Impact: Likelihood [1-5]: 5

Impact [1-5]: 5

Exploitability [1,2]: 1

Complexity [0-2]: 1

Final Score: 4.8 [Critical]

Recommendations

Recommendation:

- Distinct Variables for Deposits and Rewards:
  - Implement separate variables for tracking user deposits and rewards. This separation ensures that the calculation of rewards is solely based on the actual reward pool, unaffected by any new deposits.
- Refactoring Post-Campaign Deposit Logic:
  - Redesign the logic for deposits made after the campaign and after rewards have been distributed. These deposits should not impact the existing reward pool but rather contribute towards any new rewards added subsequently. This change ensures that users who deposit during the campaign have their reward allocation unaffected by later deposits.
- Adjustment in Withdrawal Calculations:
  - Modify the `_withdrawUser` function to calculate the withdrawal amount and rewards based on distinct variables for deposits and rewards. This adjustment ensures that the total claimed rewards do not exceed the actual reward pool.
- Documentation and Code Comments Update:
  - Update the NatSpec comments and any associated documentation to clearly explain the roles and purposes of the newly introduced variables and updated logic.

Remediation (Revised commit: 246e2c8) : The MarketplacePool contract now utilizes two new variables, `_rewardIndex` and `_rewardIndexOf`, to separately track rewards and their distribution. The `_rewardIndex` is updated whenever a reward is added to the contract. The `_rewardIndexOf` records the value of `_rewardIndex` at the time of a user's deposit. A user's reward is calculated

based on the difference between the current `_rewardIndex` and the stored value in `_rewardIndexOf` for their wallet, multiplied by the user's balance. This approach resolves the calculation discrepancies and prevents the potential locking of users' funds in the contract.

Evidences

Calculation Inconsistency in MarketplacePool Due to Shared Balance Tracking

Reproduce:

PoC steps:

- Initialize and Configure Pool: CP initializes the pool with a deposit and lock period. Configure the campaign with start time, duration, and initial max pledge.
- Campaign Deposits: Advance time to the second half of the campaign. Users (user1 and user2) deposit equal amounts.
- Verify Campaign Success: Confirm the campaign was successful and note the lock end time.
- Add Post-Campaign Rewards: After the lock period, add a specified reward amount.
- Record Initial Rewards: Log initial unclaimed rewards for user1 and user2.
- CP Withdrawal Process: Allow and execute CP's withdrawal of their collateral.
- New User Deposit and Reward Check: User3 deposits post-campaign. Check and note user3's unclaimed rewards, expected to be zero.
- Withdrawal Attempts and Observations:
  - User3's withdrawal attempt fails due to calculation issues.
  - User1 withdraws successfully.
  - User2's withdrawal fails due to insufficient rewards, demonstrating the calculation inconsistency.

Test Case:

```
it("Calculation Inconsistency in MarketplacePool Due to Shared Balance Trac
king", async () => {
const poolStart = (await ethers.provider.getBlock("latest")).timestamp;

// CP initializes the pool with a deposit and sets the lock period
await pool.connect(cp1).CPInit(ethers.utils.parseEther("90000"), 5 * MONTH)
;

// Configuring the pool campaign with a start time, duration, and initial m
ax pledge
await pool.connect(bs).configurePoolCampaign(poolStart, 31 * DAY, ethers.ut
ils.parseEther("12000"));

// Fast forward time to the second half of the campaign
await advanceTime(31 * DAY / 2);

// Users deposit during the second half of the campaign
await pool.connect(user1).depositInCampaign(ethers.utils.parseEther("5000")
);
await pool.connect(user2).depositInCampaign(ethers.utils.parseEther("500
```

See more

Results:

```
MarketplacePoolStaking
User Lock End: 1720941328
Initial Unclaimed Rewards User1: 5000000000000000000000
Initial Unclaimed Rewards User2: 5000000000000000000000
Unclaimed Rewards User3: 0
```

```
Updated Unclaimed Rewards User1: 50000000000000000000002
Updated Unclaimed Rewards User2: 50000000000000000000002
✔ Calculation Inconsistency in MarketplacePool Due to Shared Balance Tracki
ng
```

Files: MartekplacePoolStaking.Calculation.test.ts

F-2024-0629 - Inconsistent Use of Upgradeable Contracts and Incomplete Initialization in MarketplacePool - Medium

Description:

The MarketplacePool contract in the Blocksquare Marketplace Pool System demonstrates a inconsistency in its use of OpenZeppelin contracts, specifically in the mixture of upgradeable (`@openzeppelin/contracts-upgradeable`) and non-upgradeable (`@openzeppelin/contracts`) OpenZeppelin libraries.

Specifically, the contract uses OwnableUpgradeable and ERC20Upgradeable from the `@openzeppelin/contracts-upgradeable` library but also imports SafeERC20 and IERC20 from the non-upgradeable `@openzeppelin/contracts` library. Additionally, the `initialize` function initializes ERC20Upgradeable but does not initialize OwnableUpgradeable. It also uses the standard ReentrancyGuard instead of its upgradeable version:

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
// ... other imports ...

function initialize(
// ... parameters ...
) external initializer {
__ERC20_init(tokenName, tokenSymbol);
// ... other initializations ...
}
```

The mix of upgradeable and non-upgradeable contract imports can lead to unexpected behavior, particularly in the context of proxy-based upgradeability patterns. Non-upgradeable contracts are not designed to function with the state storage layout of upgradeable contracts, potentially causing issues with state variable alignment and data corruption during contract upgrades.

The primary impact is the potential for malfunctioning contract behavior and data corruption upon upgrading the contract. This could lead to loss of funds or assets, unauthorized actions, or complete contract failure.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Classification

Severity: Medium

Impact:

Likelihood [1-5]: 3

Impact [1-5]: 5

Exploitability [1,2]: 2

Complexity [0-2]: 0

Final Score: 2.5 [Medium]

Recommendations

Recommendation:

To mitigate these issues, the following steps are recommended:

- Consistent Use of Upgradeable Contracts: Replace all non-upgradeable OpenZeppelin imports with their upgradeable counterparts from the `@openzeppelin/contracts-upgradeable` library.
- Proper Initialization: Modify the `initialize` function to include the initialization of both OwnableUpgradeable and ReentrancyGuardUpgradeable.
- Updated import and initialization example:

```
import "@openzeppelin/contracts-upgradeable/token/ERC20/utils/SafeERC20Upgr
adeable.sol";
import "@openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.s
ol";
import "@openzeppelin/contracts-upgradeable/security/ReentrancyGuardUpgrade
able.sol";
// ... other upgradeable imports ...

function initialize(
// ... parameters ...
) external initializer {
__ERC20_init(tokenName, tokenSymbol);
__Ownable_init();
__ReentrancyGuard_init();
// ... other initializations ...
}
```

By ensuring consistent use of upgradeable contracts and proper initialization, the MarketplacePool contract can maintain its intended functionality and security posture, especially in the context of future upgrades.

Remediation (Revised commit: d9c5ebf) : The MarketplacePool contract now uniformly employs upgradeable versions of OpenZeppelin libraries, including SafeERC20Upgradeable, IERC20Upgradeable, and ReentrancyGuardUpgradeable. The contract's `initialize` function was updated to include the initialization of both Ownable2Step and ReentrancyGuardUpgradeable, ensuring consistent functionality and security in the context of future upgrades. The constructor was adjusted to disable initializers, aligning with best practices for upgradeable contracts.

F-2024-0658 - Incorrect Maximum Pledge Calculation in depositInCampaign Function - Medium

Description:

The `depositInCampaign` function in the MarketplacePool contract is designed to manage user deposits during a campaign. It includes a dynamic calculation of the maximum amount (`currentMaxPledge`) a user can deposit, which is supposed to increase over time, specifically during the first half of the campaign.

Identified Concerns:

- Underflow Risk: The function faces an underflow issue in the `currentMaxPledge` calculation. If `_MAX_AMOUNT_DEPOSITED` is set lower than the sum of `_cpDeposit` and `_maxPledge`, the calculation can result in an underflow. This issue could potentially block all user deposits during the first half of the campaign.
- No Deposit Limit in Second Half: Once past the midpoint of the campaign, there's no individual limit on user deposits. The only restriction is the overall campaign cap (`_MAX_AMOUNT_DEPOSITED`). This could lead to a quick fulfillment of the campaign cap.
- NatSpec Misalignment: In the `configurePoolCampaign` function the NatSpec comments imply a continuous increase in `maxPledge` throughout the campaign. However, the increase is only applicable during the first half. For the remainder of the campaign, the maximum pledge a user can make decreases with each deposit and is governed by the campaign's remaining cap.

Affected Code:

- Function `configurePoolCampaign`: No validation for `_maxPledge` to prevent underflow in `depositInCampaign`.

```
/// @param maxPledge Starting maximum amount of sBST each user can invest (
increases over time)
function configurePoolCampaign(
uint256 start,
uint256 duration,
uint256 maxPledge
) external {
...
_start = start;
_duration = duration;
_maxPledge = maxPledge;
...
}
```

- Function `depositInCampaign`: Potential underflow in `currentMaxPledge` calculation and no limit on user deposits post mid-campaign

```
function depositInCampaign(uint256 amount) external {
...
if (_start + (_duration / 2) > block.timestamp) {
uint256 currentMaxPledge = _maxPledge +
(((_MAX_AMOUNT_DEPOSITED - _cpDeposit - _maxPledge) /
(_duration / 2)) * (block.timestamp - _start));
require(
_deposited[_msgSender()] + amount <= currentMaxPledge,
"MarketplacePool: Amount exceeds current max pledge!"
);
}
...
}
```

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Classification

Severity: Medium

Impact: Likelihood [1-5]: 4

Impact [1-5]: 4

Exploitability [1,2]: 2

Complexity [0-2]: 0

Final Score: 2.5 [Medium]

Recommendations

Recommendation:

- Logic Adjustment in `depositInCampaign`: Revise the `currentMaxPledge` calculation within the `depositInCampaign` function to avoid underflow and ensure it aligns with the intended campaign behavior throughout the entire campaign duration.
- Check in `configurePoolCampaign`: Implement a check within the `configurePoolCampaign` function to prevent setting a `_maxPledge` that could lead to an underflow in the `currentMaxPledge` calculation during the first half of the campaign.
- Clarify Documentation: Update NatSpec comments to accurately describe the behavior of `maxPledge` during different phases of the campaign, ensuring users have a clear understanding of its functionality.

Remediation (Revised commit: 246e2c8) : The `configurePoolCampaign` function in the MarketplacePool contract now includes a limit on the `maxPledge` parameter, ensuring it does not exceed the difference between `_MAX_AMOUNT_DEPOSITED` and `_cpDeposit`. This adjustment prevents potential underflow issues in the `depositInCampaign` function. Additionally, the NatSpec documentation was updated for clearer understanding.

Evidences

Underflow in depositInCampaign

Reproduce:

PoC Steps:

- Initialization: Set up the MarketplacePool contract and initialize it with specific values for `_cpDeposit` and `_duration`.

- Configuration: Call `configurePoolCampaign` with a `_maxPledge` value that, when combined with `_cpDeposit`, exceeds `_MAX_AMOUNT_DEPOSITED`.
- Deposit Attempt: Try to make a deposit as a regular user during the first half of the campaign.
- Expectation: The transaction reverts due to an underflow in the calculation of `currentMaxPledge`.

```
it("Underflow in depositInCampaign", async () => {
const now = (await ethers.provider.getBlock("latest")).timestamp;
// cpWallet inits pool
await pool.connect(cp1).CPInit(ethers.utils.parseEther("90000"), 5 * MONTH)
;
// bsWallet configures pool start, duration and initial maxPledg
await pool.connect(bs).configurePoolCampaign(now, 31 * DAY, ethers.utils.pa
rseEther("12000"));

// Attempting to deposit in the campaign
// Expected to revert due to underflow in currentMaxPledge calculation
await pool.connect(user1).depositInCampaign(ethers.utils.parseEther("5000")
);
});
```

Results:

```
1) MarketplacePoolStaking
Underflow in depositInCampaign:
Error: VM Exception while processing transaction: reverted with panic code
0x11 (Arithmetic operation underflowed or overflowed outside of an unchecke
d block)
```

Files: MartekplacePoolStaking.Underflow.test.ts

F-2024-0631 - Variable Shadowing in MarketplacePool Contract - Low

Description:

The MarketplacePool contract in the Blocksquare Marketplace Pool System exhibits a variable shadowing issue due to the use of a variable name that is already defined in an inherited contract. Specifically, the contract declares a private boolean variable `_initialized`, which shadows the `_initialized` variable in the Initializable contract from the OpenZeppelin library. Additionally, the `initialize` function's parameter `owner` shadows the `owner()` function from OwnableUpgradeable.

The Initializable contract, a part of the upgradeable OpenZeppelin contracts, uses `_initialized` to track its initialization status. The redeclaration of `_initialized` in MarketplacePool can lead to confusion and potential bugs, as it may be unclear which variable is being referenced at different points in the contract:

```
bool private _initialized = false;
```

Similarly, the `initialize` function's parameter `owner` may cause confusion with the `owner()` function provided by OwnableUpgradeable, which is used to determine the current owner of the contract:

```
function initialize(
// ... other parameters ...
address owner,
// ... other parameters ...
) external initializer {
// ... initialization logic ...
}
```

Shadowing variables and function names can lead to misunderstandings about which variable or function is being accessed, potentially causing logical errors in the contract's execution. While this may not directly lead to loss of funds, it can result in unpredictable behavior and difficulties in contract maintenance and upgrades.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 5

Impact [1-5]: 2

Exploitability [1,2]: 2

Complexity [0-2]: 0

Final Score: 2.3 [Low]

Recommendations

Recommendation:

To resolve these shadowing issues, the following changes are recommended:

- Rename the `_initialized` variable in MarketplacePool to a more specific name, such as `_cpInitialized`, to clearly differentiate it from the `_initialized` variable in Initializable.
- Change the `owner` parameter in the `initialize` function to a different name, such as `poolOwner`, to avoid shadowing the `owner()` function from OwnableUpgradeable.

```
bool private _cpInitialized = false;

function initialize(
// ... other parameters ...
address poolOwner,
// ... other parameters ...
) external initializer {
__ERC20_init(tokenName, tokenSymbol);
// ... other initialization logic ...
_transferOwnership(poolOwner);
}
```

By renaming these variables and parameters, the MarketplacePool contract can avoid confusion and potential logical errors associated with variable and function shadowing, leading to clearer and more maintainable code.

Remediation (Revised commit: 246e2c8) : Renamed `_initialized` to `_poolInitialized` to avoid conflict with the Initializable contract, and changed the parameter name `owner` in the `initialize` function to `ownerWallet`, distinguishing it from the `owner()` function in OwnableUpgradeable.

F-2024-0635 - Lack of Validation for Start Time and Duration in MarketplacePool's Campaign - Low

Description:

The MarketplacePool contract in the Blocksquare Marketplace Pool System lacks necessary validations for the `start` time and `duration` parameters in the `configurePoolCampaign` function. This function is crucial for setting up investment campaigns, determining when users can deposit (`depositInCampaign`) and the conditions under which withdrawals (`withdraw`, `_withdrawCP`, `_withdrawUser`) can occur.

Currently, the `configurePoolCampaign` function does not validate whether the `start` time is set in the future (i.e., `start >= block.timestamp`) or if the `duration` meets a minimum threshold.

```
function configurePoolCampaign(uint256 start, uint256 duration, uint256 max
Pledge) external {
// ... existing checks ...
_start = start;
_duration = duration;
// ...
}
```

The absence of these checks can lead to disruptions in the investment campaign's operation, potentially allowing deposits or withdrawals outside the intended time frames.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 3

Impact [1-5]: 2

Exploitability [1,2]: 2

Complexity [0-2]: 0

Final Score: 1.8 [Low]

Recommendations

Recommendation:

It is recommended to add checks in the `configurePoolCampaign` function to ensure that:

- The `start` time is set in the future (`start >= block.timestamp`).
- The `duration` meets a minimum threshold, ensuring the campaign runs for a reasonable period.

By implementing these checks, the MarketplacePool contract can ensure that investment campaigns are configured with valid and reasonable time frames.

Remediation (Revised commit: 246e2c8) : Enhanced the `configurePoolCampaign` function in the MarketplacePool contract with additional time validations. Now, the campaign start time is verified to be set in the future, and the duration is constrained between 5 to 30 days. These updates ensure campaign configurations align with intended operational timeframes.

F-2024-0661 - Restrictive CP Wallet Collateral Withdrawal Logic in MarketplacePool Contract - Low

Description:

The withdrawal process for Certified Partners (CPs) in the MarketplacePool contract is dependent on specific conditions controlled by the contract owner. This setup introduces limitations on CPs' ability to withdraw their collateral independently.

The `_withdrawCP` function allows CPs to withdraw their collateral only under two scenarios: either the campaign was unsuccessful, or the contract owner has enabled the `_isAllowedToWithdraw` flag through the `allowExtractionOfCollateral` function.

```
// Withdrawal function for CP
function _withdrawCP() private returns (uint256) {
require(
_isAllowedToWithdraw ||
(!_isCappedReached &&
_start + _duration + 10 days <= block.timestamp)
);
// Withdrawal logic
}
```

However, the `allowExtractionOfCollateral` function can only be called by the owner after a period that is effectively double the initial lock period (`_lockEnd + _lockPeriod`), essentially prolonging the CP's ability to access their funds.

```
// Owner function to allow CP collateral extraction
function allowExtractionOfCollateral() external onlyOwner {
require(
_lockEnd > 0 && _lockEnd + _lockPeriod <= block.timestamp,
"MarketplacePool: CP lock period must end!"
);
_isAllowedToWithdraw = true;
}
```

Additionally, the `extendCPLockPeriod` function allows the owner to further extend the lock period. However, this function only adds constraints on the owner's ability to activate `allowExtractionOfCollateral` and doesn't directly affect CPs' withdrawal capabilities.

```
// Owner function to extend CP's lock duration
function extendCPLockPeriod(uint256 extendBy) external onlyOwner {
require(
_lockEnd > 0 &&
!_isAllowedToWithdraw &&
_lockEnd + _lockPeriod <= block.timestamp,
"MarketplacePool: CP lock period must end!"
);
_lockPeriod += extendBy;
}
```

In essence, the `extendCPLockPeriod` serves to further delay the point at which the owner can permit CPs to withdraw their collateral, effectively rendering the function counterproductive to its intended purpose. This design grants the owner significant control over CP funds post-successful campaigns, limiting CPs' financial autonomy and potentially affecting their operational fluidity.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 1

Impact [1-5]: 5

Exploitability [1,2]: 2

Complexity [0-2]: 0

Final Score: 1.8 [Low]

Recommendations

Recommendation:

- Autonomous Withdrawal Mechanism for CPs:
  - Revise the `_withdrawCP` function to incorporate a more flexible withdrawal mechanism for CPs. Instead of solely relying on the `_isAllowedToWithdraw` flag set by the owner, consider conditions that allow CPs to initiate withdrawals autonomously.
- Revise or Remove `extendCPLockPeriod`:
  - Reassess the `extendCPLockPeriod` function to ensure it aligns with the overall objectives of the contract. If retained, restrict its usage to within the maximum lock period permissible under `CPInit` (365 days). This ensures consistency across the contract's functions and prevents excessively prolonged lock periods.
  - If the function's utility is limited or if it complicates the contract unnecessarily, consider removing it. Simplifying the contract can enhance clarity and reduce potential misinterpretations or misuse.
- Clear Documentation and NatSpec Comments:
  - Update NatSpec comments and all related documentation to reflect any changes made to the CP withdrawal process. Ensure that the updated process is clearly documented, specifying under what conditions CPs can withdraw their collateral.

Remediation (Revised commit: 246e2c8) (Mitigated with Customer notice) : The CPs collateral is intended as a protection mechanism for the community that pledges their sBST to a specific marketplace pool. Before launching a marketplace pool campaign, the CP will state clear KPIs that they intend to reach in the specified timeframe to convince the community to pledge their sBST to their project.

As pointed out by the auditor, the contract adds limitations on CPs' ability to withdraw their collateral independently. As KPIs set forth by the CP before a marketplace pool campaign is initiated might not be on-chain conditions that would allow us to implement a mechanism for the CP to initiate withdrawals

autonomously, the implemented method is therefore the intended method of unlocking CP collateral. The risks are mitigated by assigning ownership of a marketplace pool contract to the DAO's Governance board multisig (or later on in our decentralization roadmap on-chain voting with sBST). At the end of a marketplace pool cycle, the Governance board will assess the success of the CP and set the _isAllowedToWithdraw flag safeguarding the best interests of the community.

F-2024-0662 - Owner-Controlled Liquidation of CP Collateral - Low

Description:

In the MarketplacePool contract, the `liquidateCPCollateral` function grants the contract owner the authority to liquidate the entire Certified Partner (CP) collateral. This function can be invoked only under specific conditions, mainly following a successful campaign, as indicated by `_lockEnd > 0`.

Affected Code:

```
/// @notice Triggered by owner to withdraw all of CP's collateral
function liquidateCPCollateral() external onlyOwner {
require(
_lockEnd > 0 &&
!_isAllowedToWithdraw &&
_lockEnd + _lockPeriod <= block.timestamp,
"MarketplacePool: CP lock period must end!"
);
uint256 amount = _cpDeposit;
_cpDeposit = 0;
_deposited[_cpWallet] = 0;
_totalDeposited -= amount;
IERC20(_sbstContract).safeTransfer(owner(), amount);
emit LiquidateCPCollateral(amount);
}
```

The function's design is such that liquidation can occur after the lapse of a double lock period (`_lockEnd + _lockPeriod`). This period is significant as it goes beyond the standard lock period defined for user withdrawals. Given that only the owner can trigger the `allowExtractionOfCollateral` function, there is a notable period where the owner has unilateral control over the CP's deposited funds.

This design places a considerable amount of control in the hands of the owner, especially concerning the CP's collateral. In scenarios where the double lock period has elapsed, the owner can choose to liquidate the CP's collateral, transferring all funds to their address.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 2

Impact [1-5]: 5

Exploitability [1,2]: 2

Complexity [0-2]: 0

Final Score: 2.3 [Low]

Recommendations

Recommendation:

- Re-evaluate the `liquidateCPCollateral` function's necessity and its alignment with the contract's intended use and trust model. If it is deemed essential for operational or risk management reasons, consider implementing additional safeguards or conditions to balance the power dynamics between the contract owner and CPs.
- Alternatively, consider removing or significantly restricting this function. This could involve setting stringent conditions under which liquidation is permissible.
- Ensure clear and transparent communication regarding the function's existence, purpose, and the conditions under which it can be executed. This transparency is vital for maintaining trust among all stakeholders, particularly CPs who contribute collateral to the pool.

Remediation (Revised commit: 246e2c8) (Mitigated with Customer notice) : The liquidation of CPs collateral is intended for cases where the CP has failed to deliver on the promises made to the community prior to launching their marketplace pool campaign.

As pointed out by the auditor, the liquidateCPCollateral function grants the contract owner the authority to liquidate the entire Certified Partner (CP) collateral. This function can be invoked only under specific conditions, mainly following a campaign that reached its time limit, as indicated by _lockEnd > 0.

This design places a considerable amount of control in the hands of the owner, which will be assigned to the DAO’s Governance board multisig (or later on in our decentralization roadmap on-chain voting with sBST) and in scenarios where the double lock period has elapsed, the DAO’s Governance board can choose to liquidate the CP's collateral, transferring all funds to their address and for the DAO’s Governance board to later vote and decide how to manage the confiscated collateral. By default, the voted decision is to burn this supply to decrease the circulating supply of BST and positively impacting the community.

F-2024-0659 - Checks-Effects-Interactions Pattern Violation in _deposit Function - Info

Description:

The `_deposit` function in the MarketplacePool contract is responsible for handling sBST token deposits during and after the campaign. This function performs token minting and updates state variables before calling an external transfer function.

Affected Code:

```
function _deposit(uint256 amount) private {
uint256 amountToMint = 0;
if (_msgSender() == _cpWallet) {
_cpDeposit += amount;
} else {
amountToMint = (totalSupply() == 0 || _tempsBSTBalanceThis == 0)
? amount
: (amount * totalSupply()) / _tempsBSTBalanceThis;
_mint(_msgSender(), amountToMint);
_tempsBSTBalanceThis += amount;
}
IERC20(_sbstContract).safeTransferFrom(
_msgSender(),
address(this),
amount
);
_deposited[_msgSender()] += amount;
_totalDeposited += amount;
emit Deposit(_msgSender(), amount, amountToMint);
}
```

The function interacts with an external ERC20 contract (`_sbstContract`) to transfer tokens from the user to the contract after updating state variables like `_cpDeposit`, `_tempsBSTBalanceThis`, and minting tokens.

Although the primary token intended for use in this contract is sBlocksquareToken, which is generally safe from reentrant calls, the use of other ERC20 tokens could introduce reentrancy risks.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Classification

Severity: Info

Recommendations

Recommendation:

Perform the `safeTransferFrom` call in the beginning of the `_deposit` function before updating any state variables or minting tokens according to the Checks-Effects-Interactions pattern.

Remediation (Revised commit: 246e2c8) : The `_deposit` function in the MarketplacePool contract was updated. The `safeTransferFrom` call to the

ERC20 `_sbstContract` is now executed first, prior to updating state variables or token minting.

Observation Details

F-2024-0632 - Missing Events in Key Functions - Info

Description: The MarketplacePoolProxyFactory contract lacks event emissions in few key owner-only functions: `changeImplementation`, `changeBSTStakingContract`, `changeGovernanceWallet` These functions are used to update the addresses of crucial contract dependencies – the implementation of the MarketplacePool, BST staking contract and governance wallet, respectively.

The absence of events in these functions means that there is no on-chain traceability or transparency when these addresses are updated.

Assets:

- contracts/MarketplacePoolProxyFactory.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Recommendations

Recommendation: To enhance transparency and traceability, it is recommended to emit events whenever the implementation of the MarketplacePool, BST staking contract and governance wallet addresses are updated. This will allow users and external services to monitor and react to changes.

Remediation (Revised commit: 246e2c8) : Events have been integrated into the `changeImplementation`, `changeBSTStakingContract`, and `changeGovernanceWallet` functions of the MarketplacePoolProxyFactory contract. This update ensures on-chain traceability and enhances transparency for address updates.

[F-2024-0633](#) - Missing checks for `address(0)` - Info

Description: The MarketplacePoolProxyFactory contract lacks essential validations to check for the zero address (`address(0)`) in several functions. The zero address check is a fundamental security measure in smart contracts to prevent operations involving uninitialized or default addresses.

Affected Functions:

- `constructor`: Does not validate the `implementation`, `bstStakingContract`, `rewardToken`, and `governancePool` addresses. This lack of validation could result in these critical addresses being set to the zero address.
- `createMarketplacePool`: This function lacks zero address checks for `cpWallet` and `bsWallet` parameters, potentially allowing the creation of marketplace pools with invalid addresses.
- `changeImplementation`: The function allows updating the implementation logic address without checking for the zero address, risking the misconfiguration of the contract.
- `changeBSTStakingContract`: Similar to changeImplementation, this function permits changing the BST staking contract address without validating against the zero address.
- `changeGovernanceWallet`: This function enables the modification of the governance wallet address without ensuring it is not the zero address.

Assets:

- contracts/MarketplacePoolProxyFactory.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Recommendations

Recommendation: Implement zero address validations for all address parameters in the affected functions. Ensure that addresses provided to these functions are always non-zero to maintain the integrity of contract operations.

Remediation (Revised commit: 246e2c8) : Zero address validations were implemented for critical parameters in the MarketplacePoolProxyFactory contract. This includes checks in the `constructor`, `createMarketplacePool`, `changeImplementation`, `changeBSTStakingContract`, and `changeGovernanceWallet` functions, ensuring the integrity of contract operations and preventing misconfigurations with uninitialized addresses.

F-2024-0634 - Potential Loss of Ownership Control in Contract Using Ownable - Info

Description: The MarketplacePoolProxyFactory and MarketplacePool contracts employs Ownable from OpenZeppelin for ownership management.

If ownership is mistakenly transferred, it may result in the irrevocable loss of control over the contract. Any function guarded by the `onlyOwner` modifier would become inaccessible to the original owner, effectively freezing critical administrative functionalities.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]
- contracts/MarketplacePoolProxyFactory.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Recommendations

Recommendation: Integrate Ownable2StepUpgradeable for the MarketplacePool and Ownable2Step for the MarketplacePoolProxyFactory, which implements a two-step ownership transfer process. This requires the new owner to actively accept ownership, adding an additional layer of security against accidental transfers.

Remediation (Revised commit: 246e2c8) : The MarketplacePool and MarketplacePoolProxyFactory contracts were updated to use Ownable2StepUpgradeable instead of OwnableUpgradeable. This change introduces a two-step ownership transfer process, enhancing security against unintentional ownership changes.

F-2024-0660 - Missing Error Messages in require Statements - Info

Description:

In the MarketplacePool contract, the `_withdrawCP` and `_withdrawUser` functions are critical for handling the withdrawal process for the Certified Partner (CP) and regular users, respectively. These functions contain require statements crucial for enforcing business logic and validating conditions.

The `require` statements in both functions lack descriptive error messages. This absence can make it challenging for users and developers to understand the reason for transaction failures or reverts.

Affected Functions:

- _withdrawCP:

```
function _withdrawCP() private returns (uint256) {
require(
_isAllowedToWithdraw ||
(!_isCappedReached && _start + _duration + 10 days <= block.timestamp)
);
// Rest of the function...
}
```

- _withdrawUser:

```
function _withdrawUser() private returns (uint256) {
require(
(_isCappedReached && _lockEnd <= block.timestamp) ||
(!_isCappedReached && _start + _duration + 10 days <= block.timestamp)
);
require(
balanceOf(_msgSender()) > 0,
"MarketplacePool: You need to stake sBST first."
);
// Rest of the function...
}
```

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Recommendations

Recommendation:

Include clear and informative error messages in the `require` statements. This will enhance the contract's transparency and help users understand the conditions under which their transactions may fail.

As a best practice, informative error messages should not be longer than 32 bytes in order to prevent extra gas consumption.

Remediation (Revised commit: 246e2c8) : The `_withdrawCP` and `_withdrawUser` functions in the MarketplacePool contract were updated with descriptive error messages. This enhancement provides clarity on transaction failures or revert.

F-2024-0663 - Absence of Deposit Verification in Withdraw Function - Info

Description:

In the MarketplacePool contract, the `withdraw` function allows users, including the Certified Partner (CP), to withdraw their staked sBST and any applicable rewards. However, the function lacks a preliminary check to ensure that the caller (user or CP) has an existing deposit.

```
function withdraw() public nonReentrant {
require(
_isConfigured,
"MarketplacePool: Pool needs to be configured first!"
);
uint256 toReturn = 0;
if (_msgSender() == _cpWallet) {
toReturn = _withdrawCP();
} else {
toReturn = _withdrawUser();
}
IERC20(_sbstContract).safeTransfer(_msgSender(), toReturn);
}
```

The absence of a deposit verification check in the `withdraw` function can lead to wasteful transactions where users or the CP, without a current deposit, attempt to execute the function.

Assets:

- contracts/MarketplacePool.sol [https://github.com/blocksquare/oceanpoint-contracts/]

Status: Fixed

Recommendations

Recommendation:

Implement an initial check in the `withdraw` function to verify that the caller has a positive deposit (`_deposited[_msgSender()] > 0`). This check should be placed immediately after the existing configuration check (`_isConfigured`). Such a verification step will ensure that only users with existing deposits can proceed with the withdrawal process, thereby preventing futile transaction attempts and saving gas costs.

Remediation (Revised commit: 246e2c8) : In the MarketplacePool contract, the `withdraw` function was updated to include a check ensuring that the caller has a positive deposit before proceeding with the withdrawal.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/blocksquare/oceanpoint-contracts/
Commit	bdb3d6372e35685c7bdb96d0693f9f5cc55b1a90
Whitepaper	N/A
Requirements	NatSpec
Technical Requirements	https://docs.oceanpoint.fi/for-developers/marketplace-pools; NatSpec

Contracts in Scope

./contracts/MarketplacePoolProxyFactory.sol

./contracts/MarketplacePoolProxy.sol

./contracts/MarketplacePool.sol