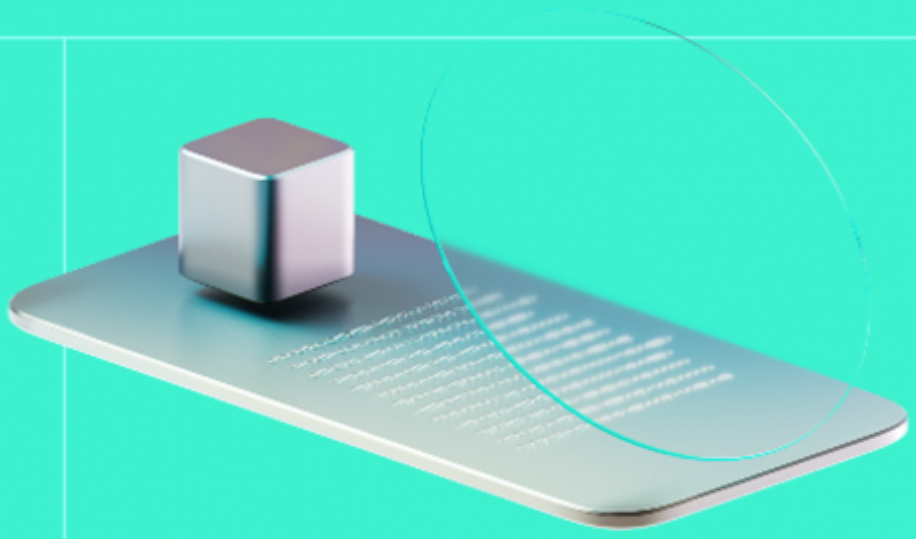
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Polybet

Date: 05.01.2024

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

Polybet is a staking platform allows tokens (different LP and PBT) staking for PBT token rewards.

Platform: EVM

Language: Solidity

Tags: ERC20 Staking

Timeline: 02.01.2024 - 05.01.2023

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/pJJ3030/jjcontracts/
Commit	bccf1b9ca9b845b85d8e3599a2c7020796e1d9db

Audit Summary

10/10	9/10	87%	10/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.3/10

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

5	0	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	0
Medium	0
Low	3

Vulnerability	Status
F-2024-0364 - PBT Token Transfer Reentrancy Possibility	Pending Fix
F-2024-0366 - LP Token Reentrancy Possibility	Pending Fix
F-2024-0370 - Lack of SafeERC20 Usage	Pending Fix
F-2024-0373 - Undistributed Staking Rewards Lock	Pending Fix
F-2024-0378 - Insufficient PBTStaking Funding Leads to User Funds Leak	Pending Fix

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 Polybet
Audited By	Stepan Chekhovskoi, Eren Gonen
Approved By	Ataberk Yavuzer
Website	https://polybet.com
Changelog	05.01.2024 - Preliminary Report

Table to Contents

System Overview

Polybet is a staking protocol with the following contracts:

PBT — simple ERC-20 token that mints all initial supply to a deployer. Additional minting is not allowed.

- Name: PBT
- Symbol: PBT
- Decimals: 18
- Total supply: 1B tokens.

PBTStaking — The PBT Staking contract enables users to stake PBT tokens and earn rewards in the form of additional PBT. It features a detailed reward mechanism based on the amount of tokens staked and the duration of staking, with a specific end block indicating the conclusion of the reward distribution period.

- The deposit function automatically compounds earned rewards for staking.
- The withdraw function withdraws all the earned rewards as well as the requested amount.

PBTLPStaking - The LP token Staking contract enables users to stake LP tokens and earn rewards in the form of PBT token. It features a detailed reward mechanism based on the amount of tokens staked and the duration of staking, with a specific end block indicating the conclusion of the reward distribution period.

- Call to deposit or withdraw functions automatically causes rewards claim.

PBTDistributor - The PBTDistributor contract is designed for token distribution using a Merkle tree to verify claims. It allows users to claim PBT tokens if their address and claim amount are verified against the immutable Merkle root stored in the contract.

Privileged roles

The system does not implement any privileged roles.

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 are provided.
- Technical description is provided.

Code quality

The total Code Quality score is **9** out of **10**.

- The PBTLPStaking and PBTStaking contracts have duplicated logic.
- The development environment is configured.

Test coverage

Code coverage of the project is **87%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is missed.
    - PBTLPStaking; constructor, withdraw and _updatePool.
    - PBTStaking; constructor, _withdraw

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **0** medium, and **3** low severity issues, 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 **9.3**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The LP token (allowed to be staked in PBTLPStaking) contract is out of the audit scope, and therefore, the security of its interactions cannot be verified.
- System owners are required to manually transfer rewards to the PBTLPStaking, PBTStaking, and PBTDistributor contracts. There is no guarantee that the contracts are able to satisfy user reward claim requests.
- PBTDistributor contract is not able to distribute funds to one user twice even if it was included in the provided Merkle tree several times.

Findings

Vulnerability Details

F-2024-0366 - LP Token Reentrancy Possibility - Low

Description:

The `PBTLPStaking` contract is designed to handle token transfers for LP tokens but do not follow the Checks-Effects-Interactions pattern in the `deposit` and `emergencyWithdraw` functions. Additionally, this contract lacks for reentrancy guards.

While the current implementation uses a pool LP token contract that is not part of the audit scope, the behavior of the LP token's transfer functionality is unknown. If any LP token contract, implements a fallback functionality on transfer (like `ERC777` do), it could introduce a reentrancy attack vector.

This oversight presents a potential security vulnerability in scenarios where the LP token contract performs external calls or interactions within its transfer functionality.

In the case, of vulnerable LP Token is set in the contract configuration:

- User funds may be griefed using the `emergencyWithdraw()` function due to the `user.amount` update happens after the `pool.lpToken.safeTransfer` interaction allowing balance double spending.
- `PBT` rewards may be stolen through the `deposit` function due to the `user.rewardDebt` update happens after the `pool.lpToken.safeTransferFrom` interaction allowing to pending rewards double spending.

```
contracts/staking/PBTLPStaking.sol: deposit(), withdraw(), emergencyWithdraw()
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Classification

Severity: Low

Impact: 4/5

Likelihood: 1/5

Recommendations

Recommendation: There are several possible solutions:

- Refactor the `deposit`, `withdraw` and `emergencyWithdraw` functions to strictly follow the Checks-Effects-Interactions pattern. Ensure that all state changes (effects) occur before external calls (interactions).
- Introduce a reentrancy guard, such as the `nonReentrant` modifier from a stable version of OpenZeppelin, to prevent reentrancy attacks.

F-2024-0373 - Undistributed Staking Rewards Lock - Low

Description:

The `PBTLPStaking` and `PBTStaking` contracts are designed for token staking, with rewards funded by project owners.

The end of the staking period is determined based on the total rewards divided by the distributed tokens per block. However, a potential issue arises if there are no depositors for a part of the staking period. In scenarios where no one stakes tokens or depositors join the pool after several blocks after the staking start, some allocated rewards would not be distributed.

```
function _updateRewards() internal {
  // Let staking be 10 blocks with 5PBT/block contract was fully funded
  // Let during 3 blocks `totalDeposits == 0`
  //                      (all funds withdrawn or nobody deposited yet)
  // Then 5PBT/block * 3blocks = 15PBT
  //                      are undistributed and locked in the contract

  ...
  uint256 denominator = totalDeposits;
  if (denominator == 0) {
    // just `lastRewardBlock` updated with corresponding rewards not distributed
    lastRewardBlock = blockToUse;
    return;
  }
  ...
}
```

This could result in a portion of the funds remaining in the contract after the staking period ends. Currently, the contracts lack a mechanism to either extend the staking period or allow the project owners to withdraw the undistributed rewards after the staking period has concluded. Consequently, this could lead to the rewards being locked in the contract permanently.

Additionally, `PBTLPStaking` uses `pool.lpToken.balanceOf(address(this))` to identify the number of staked tokens. However, the contract may hold noone owned tokens (accidentally transferred tokens, for example). The tokens also accrue rewards that cannot be withdrawn from the contract.

```
function _updatePool() internal {
  ...
  uint256 lpSupply = pool.lpToken.balanceOf(address(this));
  ...
  // Accrue reward for all the contract balance, however,
```

```
    //                                  it may include unowned funds
    pool.accPbtPerShare += pbtReward * 1e12 / lpSupply;
    ...
}
```

```
contracts/staking/PBTLPStaking.sol
contracts/staking/PBTStaking.sol
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Classification

Severity: Low

Impact: 2/5

Likelihood: 2/5

Recommendations

Recommendation: Introduce a secure function that allows the contract owner to withdraw any remaining rewards after the staking period has ended. This function should include appropriate checks to ensure it can only be called after staking period and respect funds waiting for users claim request. Fix the `PBTLPStaking` contract to accrue reward only for the staked tokens, not for the total balance.

F-2024-0378 - Insufficient PBTStaking Funding Leads to User Funds Leak - Low

Description:

Due to the reward and staking tokens being the same in the `PBTStaking` contract, if the staking is not funded with rewards or is funded insufficiently, the contract would pay rewards to the claimers, spending user funds.

The issue originates from the `_updatePool` function, which doesn't check if the contract holds enough funds to satisfy the growing `accPbtPerShare` value.

This may result in the last claimer's funds being partially distributed as rewards.

```
contracts/staking/PBTStaking.sol: _updatePool()
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Classification

Severity: Low

Impact: 2/5

Likelihood: 2/5

Recommendations

Recommendation: There are several possible solutions:

- Fully fund the contract with rewards during deployment.
- Ensure the contract `PBT` token balance doesn't fall below the `totalDeposits` value during withdrawal.
- Implement the `pbtForRewards` checking mechanism, similar to the one in the `PBTLPStaking` contract, to validate that rewards allocated don't exceed the rewards balance.

F-2024-0364 - PBT Token Transfer Reentrancy Possibility - Info

Description:

The `PBTStaking` and `PBTLPStaking` contracts are designed to handle token transfers for the `PBT` token but do not follow the Checks-Effects-Interactions pattern in the `deposit`, `withdraw` and `emergencyWithdraw` functions correspondingly. Additionally, these contracts lacks for reentrancy guards.

The current implementation uses `PBT` token contract that is part of the audit scope and prevents the reentrancy possibility. However, the mentioned attack vector may become possible during further development.

```
contracts/staking/PBTLPStaking.sol: deposit(), withdraw()
contracts/staking/PBTStaking.sol: emergencyWithdraw()
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Classification

Severity: Info

Recommendations

Recommendation:

There are several possible solutions:

- Refactor the `deposit`, `withdraw` and `emergencyWithdraw` functions to strictly follow the Checks-Effects-Interactions pattern. Ensure that all state changes (effects) occur before external calls (interactions).
- Introduce a reentrancy guard, such as the `nonReentrant` modifier from a stable version of OpenZeppelin, to prevent reentrancy attacks.

F-2024-0370 - Lack of SafeERC20 Usage - Info

Description:

`SafeERC20` is a library allows to safely interact with different partially incompatible `ERC20` tokens.

The library is unused during the interactions with known `PBT` token that is fully `ERC20` compatible and, thus, do not strongly require the library usage.

However, to prevent possible issues during further development or the project forks, the `SafeERC20` library integration is considered to be a best practice.

```
contracts/merkle/PBTDistributor.sol: claim()
contracts/staking/PBTLPStaking.sol: deposit(), withdraw()
contracts/staking/PBTStaking.sol: emergencyWithdraw(), _deposit(), _withdraw()
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Classification

Severity: Info

Recommendations

Recommendation: Integrate the `SafeERC20` library or provide comments regarding the tokens under the variables are required to be fully `ERC20` compatible.

Observation Details

F-2024-0365 - Variables Could be Marked Immutable - Info

Description: The variables assigned in the `constructor` and are never changed later could be marked `immutable`.

The modifier usage spends less Gas during the contract interaction.

```
contracts/staking/PBTStaking.sol: endBlock, pbtPerBlock
contracts/staking/PBTLPStaking.sol: pbt, pbtPerBlock
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Recommendations

Recommendation: Apply the `immutable` modifier to the mentioned variable declarations.

F-2024-0367 - Lack of Variable Visibility Modifier - Info

Description:

The `accPbtPerShare` variable visibility is not specified.

Variable visibility modifiers allows to explicitly declare the scope in which the variable can be accessed or modified.

Lack of the visibility modifier may lead to the variable being unexpectedly modified out of initial contract scope.

```
contracts/staking/PBTStaking.sol: accPbtPerShare
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Recommendations

Recommendation: Apply one of the `private`, `public`, or `internal` visibility modifiers to the variable.

F-2024-0368 - Possibly Unused Variable - Info

Description:

The `pbtForRewards` variable in the `PBTStaking` contract is only increased in the `_updateRewards` function and does not have any payload except of being viewed from off-chain.

However, based on the `pbtForRewards` variable usage in the `PBTLPStaking` contract, it could be assumed that the `pbtForRewards` variable in the `PBTStaking` contract is redundant or the contract is not finalized.

```
contracts/staking/PBTStaking.sol: pbtForRewards
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Recommendations

Recommendation:

Remove the `pbtForRewards` variable from the code if it is unneeded, or provide declarative comments on its purpose.

F-2024-0369 - Inefficient Gas Usage Due to Excessive Error Message Length - Info

Description:

In the `PBTLPStaking` contract, the `_updatePool` function contains a require statement with an error message that exceeds the 32-byte threshold. The specific require statement in question is:

```
require(pbt.balanceOf(address(this)) - pbtForRewards >= pbtReward,
                                    "Insufficient PBT tokens for rewards");
```

In Solidity, error messages that exceed 32 bytes lead to higher gas costs. This is because longer error messages consume more gas to store and process. The current error message, `"Insufficient PBT tokens for rewards"`, is longer than 32 bytes and, thus, contributes to inefficient gas usage.

```
contracts/staking/PBTLPStaking.sol: _updatePool()
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Recommendations

Recommendation:

Revise the error message in the mentioned `require` statement to be within the 32-byte limit. For example, the message could be shortened to simply `"Insufficient rewards"`.

F-2024-0371 - Staking Contracts Logic Duplication - Info

Description:

The `PBTLPStaking` and `PBTStaking` contracts share a significant amount of common logic and structure.

The duplication is particularly noticeable in the handling of user information, reward calculations, and the update of reward variables. Both contracts define similar `UserInfo` structure, use similar state variables like `lastRewardBlock`, `endBlock`, `accPbtPerShare`, and implement analogous functions `_updateRewards`, `deposit`, `withdraw`, `emergencyWithdraw`.

This duplication leads to increased code size, reduced readability, and potential challenges in further contracts development.

```
contracts/staking/PBTLPStaking.sol
contracts/staking/PBTStaking.sol
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Recommendations

Recommendation:

Develop an abstract contract that encapsulates the shared logic and structures between the `PBTLPStaking` and `PBTStaking` contracts. This abstract contract should include common functionalities like reward calculations, user information handling, and reward updates.

F-2024-0372 - Allowance of Deposits After Staking Period - Info

Description:

The `PBTLPStaking` and `PBTStaking` contracts currently permit users to deposit tokens even after the staking period has concluded (i.e., after the `endBlock` has been reached).

```
function deposit(uint256 _amount) external {
  PoolInfo storage pool = poolInfo;
  // Lack of `require(pool.endBlock > block.number)` check
  ...
}
```

This functionality presents a logical inconsistency, as users are able to deposit tokens into the staking contract without the possibility of earning any further rewards once the staking period is over.

Allowing deposits post the `endBlock` does not align with the typical purpose of a staking contract, which is to incentivize token holding during a specified period with rewards.

```
contracts/staking/PBTLPStaking.sol: deposit()
contracts/staking/PBTStaking.sol: deposit()
```

Assets:

- Polybet [https://github.com/pJJ3030/jjcontracts]

Status: Pending Fix

Recommendations

Recommendation:

Modify the deposit function in both the `PBTLPStaking` and `PBTStaking` contracts to include a check that prevents deposits if the staking end block has been reached or passed.

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/pJJ3030/jjcontracts/
Commit	bccf1b9ca9b845b85d8e3599a2c7020796e1d9db
Whitepaper	N/A
Requirements	README.md
Technical Requirements	README_Staking.md

Contracts in Scope

./contracts/PBTDistributor.sol

./contracts/PBT.sol

./contracts/PBTLPStaking.sol

./contracts/PBTStaking.sol