QuillAudits

Audit Report, July, 2024

For

PETO BOTS

Table of Content

Table of Content

Executive Summary

Project Name	Petobots
Overview	Petobots is an NFT-based GameFi project where players collect robotic pets and use them to clash in PvP and PvE battles. Games are available both with wagers and free-to-play mode. Players need a Petobot NFT to receive an in-game character that can be leveled up and improved. However, it is possible to test the gameplay with a default character by simply connecting with MetaMask or WalletConnect.
Method	Manual Review, Functional Testing, Automated Testing.
Audit Scope	The scope of this audit was to analyze Staking codebase for quality, security, and correctness.
Source Code	StakingContractBlast (WETH), mainnet: https://blastscan.io/address/0x00aa056a05A8e67602dE6F090f072089541E2678#code StakingContractBlast (USDB), mainnet: https://blastscan.io/address/0xe5576952F2D7927A37C2d7089b19c7Bb4586DA01#code
Timeline	26th June 2024 - 23rd July 2024
Updated Code Received	19th July 2024
Final Review	20th July 2024 - 23rd July 2024
Fixed In	https://github.com/cryptoalmaspace/staking-contract-audit
Mainnet Address	StakingContractBlast (WETH): 0xeaC59270630848F0585db40dFCB808f0D1FBe995 https://blastscan.io/address/0xeaC59270630848F0585db40dFCB808f0D1FBe995#code StakingContractBlast (USDB): 0x9930D2f207a68C7DA36387C164B4f8B40A72A513 https://blastscan.io/address/0x9930D2f207a68C7DA36387C164B4f8B40A72A513#code

Number of Issues per Severity

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	0	3
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	3	7	2

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Staking contract

  Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Hardhat, Solhint, Slither, Mythril, Solidity statistical analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

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

Gas-Optimisation

Gas optimisation techniques are changes which can help in reducing the gas required

Best-Coding practices

These are suggestions for better code readability and consistency
https://docs.soliditylang.org/en/v0.8.14/style-guide.html

Confirmational

These are some comments which are added by us just to confirm the intention of a piece of code.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

A. Contract - StakingContractBlast contract

High Severity Issues

No issues were found.

Medium Severity Issues

A.1 Inefficient storage and retrieval of accounts and deposit IDs with scope of DoS attack

Line **Function -**

127

```
function _accountExists(address account) private view returns (bool) {
    bool found = false;
    for (uint I = 0; I < _accounts.length; ++i) {
      if (_accounts[I] == account) {
        found = true;
        break;
      }
    }
    return found;
  }

  /**
   * Get all Account's Deposits.
   *
   * @param account Account for which information is needed
   */
  function getAccountDeposits(address account) public view returns
(DepositInfo[] memory) {
    uint depositsCount = _depositCounters[account];
    if (depositsCount == 0) {
      DepositInfo[] memory resNull;
      return resNull;
    }

    DepositInfo[] memory res = new DepositInfo[](depositsCount);
    uint ii = 0;
    for (uint I = 0; I < _depositIds.length; ++i) {
      if (_deposits[_depositIds[I]]._owner == account) {
        res[ii] = _deposits[_depositIds[I]];
        ++ii;
      }
    }

    return res;
```

Description

The code currently uses arrays to store _accounts and _depositIds, which leads to inefficient operations such as checking if an account exists (_accountExists function) and retrieving all deposits for an account. This inefficiency can lead to a Denial of Service (DoS) attack when the length of _accounts becomes large, causing functions that loop through the array to consume excessive gas and potentially fail.

Remediation

- Replace the _accounts and _depositIds arrays with Enumerable from OpenZeppelin.
- Update functions to utilize these sets for efficient storage and retrieval, reducing the risk of DoS attacks due to gas limitations.

Status

Resolved

A.2 Probable Hash collisions in deposit ID generation using abi.encodePacked

Line Function -

```
232     bytes32 depositId = bytes32(keccak256(abi.encodePacked(_msgSender(),
        block.number)));
            // regenerate ID
            if (_deposits[depositId]._id != 0) {
              depositId = bytes32(keccak256(abi.encodePacked(depositId)));
            }

270
        emit Deposit(
            sender,
            string(abi.encodePacked(depositId)),
            amount,
            currentTime,
            dep._availableFrom
          );

309
        emit Withdraw(sender, string(abi.encodePacked(depositId)), amount,
        getCurrentTime());
```

Description

The code uses abi.encodePacked for generating the depositId, which might lead to hash collisions. Using abi.encode is more appropriate in this context.

Remediation

• Replace abi.encodePacked with abi.encode when generating depositId.
• Ensure no hash collisions and test thoroughly.

OR

• Use a _deposits_counter instead of block.number to generate unique deposit IDs.

Status

Resolved

A.3 Scope of Reentrancy in withdraw function

Line Function -

232 bytes32 depositId = bytes32(keccak256(abi.encodePacked(_msgSender(), block.number)));

```
    // regenerate ID
    if (_deposits[depositId]._id != 0) {
      depositId = bytes32(keccak256(abi.encodePacked(depositId)));
    }
```

Description

The _token.transfer call should be moved to the end of the withdraw function to follow best practices and avoid reentrancy issues. The nonreentrant guard is used but still there might be a global reentrancy and it is best practice to follow check effect interaction pattern.

Remediation

Move _token.transfer call to the end of the withdraw function after all state changes are made.

Status

Resolved

A.4 Risk of centralization in blast functions

Description

Contracts have owners with privileged rights to perform admin tasks and need to be trusted to not perform malicious updates or drain funds. The functions configureBlastPoints, configureBlastYieldModes, claimYieldAll, and claimGas have significant control and could be exploited if not properly secured.

Remediation

- Implement multi-signature requirements for critical functions to reduce the risk of a single point of failure.
- Consider using a decentralized governance model to manage administrative privileges.

Status

Acknowledged

Low Severity Issues

A.5 Removal of redundant amount parameter

Line **Function -**

172

```
function deposit(uint256 amount, uint8 periodType) external whenNotPaused
nonReentrant {
    address sender = _msgSender();


    require(periodType < 3, “periodType must be less than 3”);
    require(
      _token.allowance(sender, address(this)) >= amount,
      “You must allow to use of funds by the Contract”
    );
    require(_token.balanceOf(sender) >= amount, “You don’t have enough
funds”);


    if (periodType == 0) {
      require(amount == _period0Price, “Amount must be equal to
_period0Price”);
      if (_period0MaxAmount > 0) {
        require(
          (_period0AmountsCounter + amount) <= _period0MaxAmount,
          “The limit of the Period-0 has been reached”
        );
      }
      if (_period0MaxAmountPerAccount > 0) {
        require(
          (_period0AmountCounters[sender] + amount) <=
_period0MaxAmountPerAccount,
          “You have reached the limit per account (Period-0)”
        );
      }
    } else if (periodType == 1) {
      require(amount == _period1Price, “Amount must be equal to
_period1Price”);
      if (_period1MaxAmount > 0) {
        require(
```

```
            (_period0AmountsCounter + amount) <= _period0MaxAmount,
            “The limit of the Period-0 has been reached”
          );
        }
        if (_period0MaxAmountPerAccount > 0) {
          require(
            (_period0AmountCounters[sender] + amount) <=
_period0MaxAmountPerAccount,
            “You have reached the limit per account (Period-0)”
          );
        }
      } else if (periodType == 1) {
        require(amount == _period1Price, “Amount must be equal to
_period1Price”);
        if (_period1MaxAmount > 0) {
          require(
            (_period1AmountsCounter + amount) <= _period1MaxAmount,
            “The limit of the Period-1 has been reached”
          );
        }
        if (_period1MaxAmountPerAccount > 0) {
          require(
            (_period1AmountCounters[sender] + amount) <=
_period1MaxAmountPerAccount,
            “You have reached the limit per account (Period-1)”
          );
        }
      } else if (periodType == 2) {
        require(amount == _period2Price, “Amount must be equal to
_period2Price”);
        if (_period2MaxAmount > 0) {
          require(
            (_period2AmountsCounter + amount) <= _period2MaxAmount,
            “The limit of the Period-2 has been reached”
          );
        }
        if (_period2MaxAmountPerAccount > 0) {
          require(
            (_period2AmountCounters[sender] + amount) <=
```

```
                _period2MaxAmountPerAccount,
                    "You have reached the limit per account (Period-2)"
                );
            }
        }
```

Description

The amount parameter in the deposit function is redundant as the amount is implicitly defined by the periodType.

Remediation

- Remove the amount parameter from the deposit function.
- Update the require statements to check against the appropriate period price based on periodType.

Status

Resolved

A.6 Emit event for updateDepositDuration function

Line Function -

317 function updateDepositDuration(bytes32 depositId) external {
address sender = _msgSender();

DepositInfo memory dep = _deposits[depositId];
require(dep._owner == sender, “You’re not the Owner of the Deposit”);

uint64 duration = 0;
if (dep._periodType == 0) {
duration = _period0Duration;
} else if (dep._periodType == 1) {
duration = _period1Duration;
} else if (dep._periodType == 2) {
duration = _period2Duration;
}

_deposits[dep._id]._availableFrom = _deposits[dep._id]._createTime +
duration;
}

Description

The updateDepositDuration function does not emit an event, making it difficult to track changes to deposit durations.

Remediation

Define and emit an event when the deposit duration is updated in the updateDepositDuration function.

Status

Resolved

A.7 Emit event for setPeriodDuration function

Line Function -

341
```
function setPeriodDuration(uint8 periodType, uint64 newDuration) external
onlyOwner {
    if (periodType == 0) {
      require(
        newDuration < _period0Duration,
        “newDuration must be less than current _period0Duration”
      );
      _period0Duration = newDuration;
    } else if (periodType == 1) {
      require(
        newDuration < _period1Duration,
        “newDuration must be less than current _period1Duration”
      );
      _period1Duration = newDuration;
    } else if (periodType == 2) {
      require(
        newDuration < _period2Duration,
        “newDuration must be less than current _period2Duration”
      );
      _period2Duration = newDuration;
    }
  }
```

Description

The setPeriodDuration function does not emit an event, making it difficult to track changes to period durations.

Remediation

Define and emit an event when the period duration is updated in the setPeriodDuration function.

Status

Resolved

A.8 Emit deposit ID as bytes32 in Withdraw event

Line	Function -
309	emit Withdraw(sender, string(abi.encodePacked(depositId)), amount, getCurrentTime());

Description

The Withdraw event currently emits depositId as a string, which should be emitted as bytes32 for better consistency and tracking.

Remediation

Modify the Withdraw event to emit depositId as bytes32.

Status

Resolved

A.9 Pragma version not locked

Line	Function -
1	// SPDX-License-Identifier: MIT pragma solidity ^0.8.23;

Description

The pragma version is currently set to ^0.8.23, which allows for any minor version above 0.8.23 to be used for compilation. This can introduce unexpected behavior due to changes in the compiler.

Remediation

Lock the pragma version to 0.8.23 to ensure consistent behavior across compilations.

Status

Resolved

A.10 Catch return value from _USDB.configure(IERC20Rebasing.YieldMode.CLAIMABLE) And _WETHB.configure(IERC20Rebasing.YieldMode.CLAIMABLE)

Line Function -

379

```
function configureBlastPoints(
    address blastUsdbYieldAddress,
    address blastWethbYieldAddress,
    address blastPointsAddress,
    address blastPointsOperator
  ) public onlyOwner {
    _BLAST.configureClaimableGas();
    _USDB = IERC20Rebasing(blastUsdbYieldAddress);
    _USDB.configure(IERC20Rebasing.YieldMode.CLAIMABLE);
    _WETHB = IERC20Rebasing(blastWethbYieldAddress);
    _WETHB.configure(IERC20Rebasing.YieldMode.CLAIMABLE);


    _blastPointsAddress = blastPointsAddress;
    _blastPointsOperator = blastPointsOperator;

IBlastPoints(_blastPointsAddress).configurePointsOperator(_blastPointsOperator)
;
  }
```

Description

The return value from _USDB.configure(IERC20Rebasing.YieldMode.CLAIMABLE) is currently ignored, which can lead to missed errors or unsuccessful configurations.

Remediation

Assign the return value to a variable and handle it appropriately to ensure the function executed successfully.

Status

Resolved

A.11 Remove unnecessary id field from DepositInfo struct

Description

The _id field in the DepositInfo struct is not needed and adds unnecessary complexity to the structure.

Remediation

Remove the _id field from the DepositInfo struct to simplify the structure.

Status

Resolved

Informational Issues

A.12 Optimize gas usage by deleting deposit information on withdraw

Line	Function -
307	_deposits[dep._id]._withdrawn = true;

Description

The _deposits mapping retains withdrawn deposit information by setting _withdrawn to true instead of deleting the entry, leading to higher gas costs.

Remediation

- Delete the entry from _deposits mapping upon withdrawal to optimize gas usage.
- Ensure that any necessary information is logged or stored elsewhere before deletion.

Status

Acknowledged

A.13 Notation of storage and immutable variables

Description

Prefixing storage variables with s_ and immutable variables with i_ improves readability and clearly distinguishes these variables from others, making it evident which variables are stored in the contract's state and which are immutable.

Status

Acknowledged

A.14 Name Mappings

Description

Named mappings can enhance code readability and understanding. For instance, instead of using generic names, use descriptive names that indicate the purpose of the mapping.

Status

Acknowledged

A.15 Confirm if withdraw function should be pausable?

Description

The withdraw function is not protected by the whenNotPaused modifier. This could allow withdrawals even when the contract is paused.

Remediation

- Add the whenNotPaused modifier to the withdraw function if withdrawals should be paused when the contract is paused.
- Confirm if the contract's behavior is intended to allow withdrawals while paused.

Status

Resolved

A.16 Confirm if _allDepositsAmountCounter should decrement on withdraw?

Description

The _allDepositsAmountCounter does not decrement when a deposit is withdrawn, potentially leading to incorrect total deposit amount tracking.

Remediation

- Decrement _allDepositsAmountCounter by the withdrawn amount in the withdraw function.
- Confirm if this behavior aligns with the intended functionality of the contract.

Status

Resolved

Functional Tests Cases

Some of the tests performed are mentioned below:

Contract : Fund

- ✓ User makes Deposits (in USDB)
- ✓ User makes a Withdrawal (in WETH)

Test Coverage

Test coverage for the contracts is Good.

all files contracts/

97.62% Statements 82/84 **68.37%** Branches 67/98 **100%** Functions 27/27 **97.18%** Lines 138/142

File	Statements		Branches		Functions		Lines
StakingContractBlast.sol	97.56%	80/82	68.75%	66/96	100%	20/20	97.12%
TestUSD.sol	100%	2/2	50%	1/2	100%	7/7	100%

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Petobots StakingContractBlast contract. We performed our audit according to the procedure described above.
Some issues of Medium and Low severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

As Recommended Petobots Team Resolved all Issues. The code does follow the solidity style guide thoroughly. And finally the test branch coverage is almost 98 percent.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in Petobots StakingContractBlast smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of Petobots StakingContractBlast smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the Petobots StakingContractBlast to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
July, 2024

For

PETO BOTS

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com