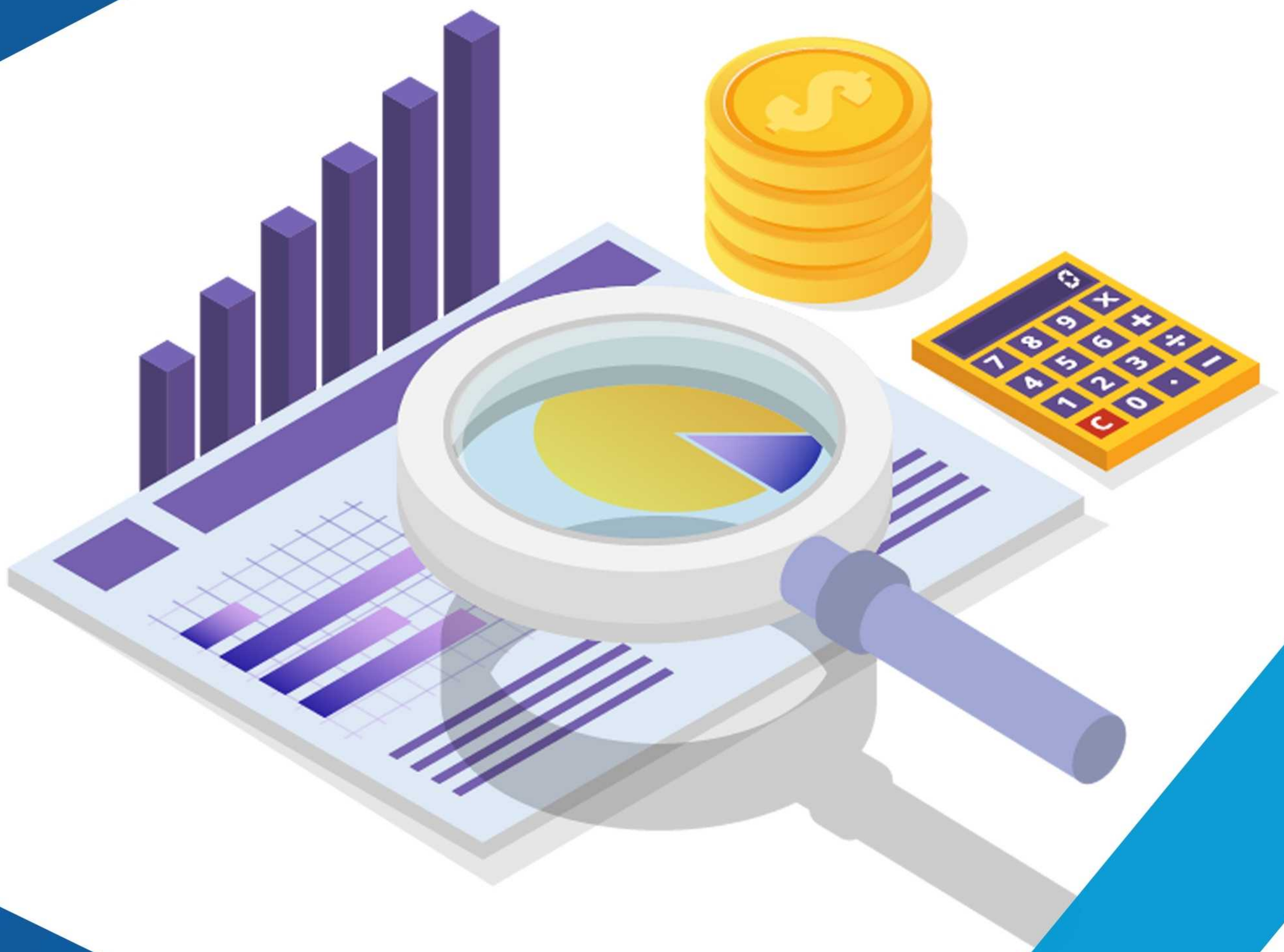
QuillAudits



Audit Report
August, 2021

PROXY

Contents

Scope of Audit	01
Techniques and Methods	02
Issue Categories	03
Issues Found – Code Review/Manual Testing	04
Automated Testing	12
Disclaimer	13
Summary	14

Scope of Audit

The scope of this audit was to analyze and document the btcproxy Token smart contract codebase for quality, security, and correctness.

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- 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
- Malicious libraries
- 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 contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- 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.
SmartCheck.

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

Code Review / Manual Analysis
Manual Analysis or review of code was done to identify new vulnerability 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 automated analysis were manually verified.

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

Tools and Platforms used for Audit
Remix IDE, Truffle, Truffle Team, Ganache, Solhint, Mythril, Slither, SmartCheck.

Issue Categories

Every issue in this report has been assigned with 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 to be fixed before moving to a live environment.

Medium level 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 level severity issues

Low level severity issues can cause minor impact and or 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 severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	2	0	0	0
Acknowledged	0	0	3	3
Closed	3	0	0	2

Introduction

During the period of **Aug 8, 2021 to Aug 15, 2021** - QuillAudits Team performed a security audit for btcproxy smart contracts.

The code for the audit was taken from the following official link: https://github.com/Proxy-Protocol/Rewards-Staking

Note	Date	Commit hash
Version 1	August	9cbb7d7b00b5c5ab2cb2417d1081a83dcc9cdf84
Version 2	August	c7bef8e92eba31299597e0448e6fd7663b527266
Version 3	August	c3d6de1321ce34975f9f417a87a76e1be4d43281

Issues Found – Code Review / Manual Testing

High severity issues

1. stake function might get invokable
   Line No: 120

```
function stake(uint256 amount) external updateReward(msg.sender) {
    require(amount > 0, "Cannot stake 0");
    _totalSupply = _totalSupply.add(amount);
    //decimal till 6 places for %
    //dividing apy by 6 decimal to get percentage in whole number
    //if apy whole number percenatge is less than 10%
    if(getAPY().div(1e6) <= 10) {
        if(lastTimeApyLessThan10 == 0) {
            lastTimeApyLessThan10 = block.timestamp;
        } else if(block.timestamp > lastTimeApyLessThan10.add(1 days) && !apyLessThan10) {
            lastTimeApyLessThan10 = block.timestamp;
        } else if(block.timestamp > lastTimeApyLessThan10.add(1 days)) {
            revert("Err: APY is less than 10%");
        }
        apyLessThan10 = true;
    } else {
        if(block.timestamp > lastTimeApyLessThan10.add(1 days)) {
            revert("Err: APY is less than 10%");
        }
        lastTimeApyLessThan10 = block.timestamp;
        apyLessThan10 = false;
    }
    _balances[msg.sender] = _balances[msg.sender].add(amount);
    IERC20(stakeToken).safeTransferFrom(msg.sender, address(this), amount);
    emit Staked(msg.sender, amount);
}
```

Description

The function stake() is updating lastTimeApyLessThan10 in all conditions, it’s possible to have a delay between two calls to staking of more than a day. In such cases, stake() will always revert until apy gets less than 10.
Another case, if apy gets below 10 & lastTimeApyLessThan10 is updated and there is a time of 1 day in next stake function call, then stake() will always be reverting even if apy gets more than 10.

Remediation

Analyze all the conditions in which lastTimeApyLessThan10 needs to be updated, analyze dead ends and write unit tests to assert such conditions.

Status: Closed

This has been fixed in commit c7bef8e92eba31299597e0448e6fd7663b527266

2. Contract balance is not checked before adding rewards
Line No: 169

```
function notifyRewardAmount(uint256 reward)
    external
    onlyRewardDistribution
    updateReward(address(0))
{
    if (block.timestamp >= periodFinish) {
        rewardRate = reward.mul(1e6).div(stakingDuration());
    } else {
        uint256 remaining = periodFinish.sub(block.timestamp);
        uint256 leftover = remaining.mul(rewardRate);
        rewardRate = reward.mul(1e6).add(leftover).div(stakingDuration());
    }
    lastUpdateTime = block.timestamp;
    periodFinish = stakingStartDate.add(stakingDuration());
    rewardToBeDistributed = reward;
    emit RewardAdded(reward);
}
```

Description
In notifyRewardAmount function, the respective balance for the reward being added is not checked, reward distributor maliciously can vouch to add millions of tokens but at the time of reward payout functions will revert.

Remediation
There should be a conditional check to validate the reward amount, this will restrict malicious reward distributors to add imaginary reward.

Status: Open

3. periodFinish value might get inaccurate
Line No: 169

```
function notifyRewardAmount(uint256 reward)
    external
    onlyRewardDistribution
    updateReward(address(0))
{
    if (block.timestamp >= periodFinish) {
        rewardRate = reward.mul(1e6).div(stakingDuration());
    } else {
        uint256 remaining = periodFinish.sub(block.timestamp);
        uint256 leftover = remaining.mul(rewardRate);
        rewardRate = reward.mul(1e6).add(leftover).div(stakingDuration());
    }
    lastUpdateTime = block.timestamp;
    periodFinish = stakingStartDate.add(stakingDuration());
    rewardToBeDistributed = reward;
    emit RewardAdded(reward);
}
```

Description

In the notifyRewardAmount function, the periodFinish value might get inaccurate. The value is being updated on the basis of the start date of staking and time left for staking, which is not correct. Let's take an example, let's say staking started at 10 units of time and is gonna be finished at 200 units of time. And for the first call to notifyRewardAmount, periodFinish will be 200 units of time. Reward distributor wants to increase reward at 150 units of time, he added some reward, periodFinish now set at 70 units of time. Now if the reward distributor adds additional reward, he will be replacing the previous reward rate instead of updating it.

Remediation

periodFinish calculation should be corrected.

Status: Closed

This has been fixed in commit c3d6de1321ce34975f9f417a87a76e1be4d43281 by making notifyRewardAmount inaccessible.

4. rewardsToBeDistributed value might get inaccurate
Line No: 169

```
function notifyRewardAmount(uint256 reward)
    external
    onlyRewardDistribution
    updateReward(address(0))
{
    if (block.timestamp >= periodFinish) {
        rewardRate = reward.mul(1e6).div(stakingDuration());
    } else {
        uint256 remaining = periodFinish.sub(block.timestamp);
        uint256 leftover = remaining.mul(rewardRate);
        rewardRate = reward.mul(1e6).add(leftover).div(stakingDuration());
    }
    lastUpdateTime = block.timestamp;
    periodFinish = stakingStartDate.add(stakingDuration());
    rewardToBeDistributed = reward;
    emit RewardAdded(reward);
}
```

Description

In the notifyRewardAmount function, the rewardsToBeDistributed value might get inaccurate. On subsequent increments of reward, the rewardsToBeDistributed is getting replaced by the additional reward rather than getting updated.

Remediation

rewardsToBeDistributed should reflect additional reward too, it shouldn't be replaced each time a new reward is added.

Status: Closed

This has been fixed in commit c3d6de1321ce34975f9f417a87a76e1be4d43281 by making notifyRewardAmount inaccessible.

5. rewardsToBeDistributed value might get inaccurate
Line No: 169

```
function recoverExcessToken(address token, uint256 amount) external onlyOwner {
    IERC20(token).safeTransfer(_msgSender(), amount);
    emit RecoverToken(token, amount);
}
```

Description
In the recoverExcessToken function, the owner will be able to steal the reward token added by the reward distributor.

Remediation
Reward tokens shouldn't be withdrawable by the owner at any time, have a set of conditions under which this can happen, and also add documentation and provide specifications for users regarding it. Reflect the changes in reward calculation computation too.

Status: Open

Low level severity issues

6. Lack of zero address check

Description
Across the codebase, there is a lack of zero address check for functionalities such as administrative and initialization.

Remediation
Either have zero address checks within the contract itself or have robust deployment scripts to check for these scenarios.

Status: Acknowledged

7. Do not use SafeMath library

Description
Solidity compiler v0.8.4 is being used for compilation. The compiler already has native protections against arithmetic underflow and overflow.

Remediation
Do not use Safemath library with V0.8.4 compiler, it just wastes gas.

Status: Acknowledged

8. Lack of unit tests

Description
There are no unit tests for the codebase, the coverage is zero. Contracts that are meant to handle thousands of user funds should have appropriate testing and coverage.

Remediation
Implement unit tests with at least 98% coverage.

Status: Acknowledged

Informational

9. Lack of documentation on scaling of decimals of reward token
Line No: 202

```
function getAPY() public view returns (uint256) {
    //3153600000 = 365*24*60*60
    if(block.timestamp > stakingEndDate){
        return 0;
    }
    uint256 rewardForYear = rewardRate.mul(31536000);

    address[] memory path = new address[](3);
    path[0] = stakeToken;
    path[1] = WETH;
    path[2] = USDT;

    uint256 _stakeTokenUSD = getOutputAmount(1 * 10 ** (IERC20Metadata(stakeToken).decimals()), path);

    path[0] = WBTC;
    uint256 _rewardTokenUSD = getOutputAmount(1 * 10 ** (IERC20Metadata(rewardToken).decimals()), path);

    //multiplied by 1e10 to match the decimals with prxy decimals
    //% till 6 decimal places
    return ((((rewardForYear.mul(1e10).mul(_rewardTokenUSD)).mul(1e8)).div(_totalSupply.mul(_stakeTokenUSD))).div(1e6)); // put 6 dp
}
```

Description
Throughout the codebase, the reward tokens are being scaled up and down by six decimals. This is being done to minimize token loss while calculating rewards per unit of staking token, while the implementation is correct, there is a lack of information necessary for describing it to users.

Remediation
It is recommended to write about reasons for scaling into the code, comments can be used and it won't bear any overhead gas expenses. Reward token is of 8 decimals, the scaling should be done in 8 decimals only to extremely minimize any loss.

Status: Closed
The scaling is being modified to use 8 decimals instead of 6 a recommended in commit c7bef8e92eba31299597e0448e6fd7663b527266.

10. **Possible gas optimization in storage**
Line No: 28

```
bool public apyLessThan10;
```

Description
Data type address can be arranged with data type bool to save gas.

Status: Acknowledged

11. **Do not reinitialize storage variables**
Line No: 21

```
uint256 public periodFinish = 0;
uint256 public rewardRate = 0;
```

Description
Avoid initialization of storage variables to default values, it wastes gas.

Status: Acknowledged

12. **Floating Pragma**

pragma solidity ^0.8.4;

Description
Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation
Lock the pragma version and also consider known bugs for the compiler version that is chosen.

Status: Closed
The scaling is being modified to use 8 decimals instead of 6 as recommended in commit c3d6de1321ce34975f9f417a87a76e1be4d43281.

13. **State variables that could be declared immutable**

    address public stakeToken;

    Description
    The above constant state variable should be declared immutable to save gas.

    Remediation
    Add the immutable attributes to state variables that never change after contact creation.

    Status: Acknowledged

Automated Testing

Slither

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

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the btcproxy platform. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the btcproxy team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Closing Summary

Overall, in the initial audit, there were few high-level issues associated with funds calculation, most of them are fixed now. Still, two high severity issues persist, #2 and #5, they are not fixed due to btcproxy's business requirement.

No instances of Integer Overflow and Underflow vulnerabilities or Back-Door Entry were found in the contract, but relying on other contracts might cause Reentrancy Vulnerability.

Numerous issues were discovered during the initial audit. Some of them are now fixed and checked for correctness.

PROXY

QuillAudits

Canada, India, Singapore and United Kingdom

audits.quillhash.com

audits@quillhash.com