BLOCKSEC

Security Audit Report for PumpBTC Contracts

Date: June 19, 2024 **Version:** 1.1

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	PumpBTC
Target	PumpBTC Contracts

Version History

Version	Date	Description
1.0	June 13, 2024	First release
1.1	June 19, 2024	Add a new commit hash

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-06-19 06:45:59+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The focus of this audit is on the PumpBTC Contracts of PumpBTC [1]. PumpBTC Contracts allows users to stake Wrapped Bitcoin tokens (e.g. BTCB, WBTC, FBTC) into the PumpStaking contract and mint pumpBTC tokens at a 1:1 ratio. These staked assets will be withdrawn and unwrapped into BTC to stake and earn rewards on Babylon. For unstake requests, the protocol offers standard and instant options with fees.

Please note that only the contracts located within the contracts folder in the repository are included in the scope of this audit. Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (Version 1), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
PumpBTC Contracts	Version 1	b1481f38f7f99342c6c607f1354c26a75112d4a7
	Version 2	e4960e4edda06c5bf5375b5648ad7d3ce4c73cfd
	Version 3	88c08a53a3b3cf7b753d2e26d519f4936a3047c7

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit can-

[1] https://github.com/pumpbtc/pumpBTC-contract

not be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Permission management
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact	*High*	High	Medium
	Low	Medium	Low
		High	*Low*
		Likelihood	

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **one** potential security issue. Besides, we have **three** recommendations and **three** notes.

- High Risk: 1
- Recommendation: 3
- Note: 3

ID	Severity	Description	Category	Status
1	High	Potential precision loss in the `stake` function	DeFi Security	Fixed
2	-	Remove redundant code	Recommendation	Acknowledged
3	-	Add checks on the new staking limit	Recommendation	Fixed
4	-	Follow CEI pattern in the `PumpStaking` contract	Recommendation	Fixed
5	-	Potential precision loss in the `unstakeInstant` function	Note	-
6	-	About the off-chain logic	Note	-
7	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential precision loss in the `stake` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `PumpStaking` contract, a precision loss issue exists in the `stake` function. Specifically, in this function, the `pumpBTC` amount to be minted is accepted, and the `_adjustAmount` function is used to calculate the required deposited assets. However, this result may be rounded down to zero, allowing users to mint `pumpBTC` without depositing assets. By exploiting this precision loss issue, attackers can arbitrarily mint `pumpBTC` tokens. Once the `PumpStaking` contract holds assets, attackers can then unstake to drain assets from it.

```
function stake(uint256 amount) public whenNotPaused {
    require(amount > 0, "PumpBTC: amount should be greater than 0");
    require(
        totalStakingAmount + amount <= totalStakingCap,
        "PumpBTC: exceed staking cap"
    );

    asset.safeTransferFrom(_msgSender(), address(this), _adjustAmount(amount));
    pumpBTC.mint(_msgSender(), amount);

    totalStakingAmount += amount;
```

```
        pendingStakeAmount += amount;

        emit Stake(_msgSender(), amount);
    }
```

Listing 2.1: contracts/PumpStaking.sol

```
    function _adjustAmount(uint256 amount) public view returns (uint256) {
        return assetDecimal > 18 ? amount * 10 ** (assetDecimal - 18) : amount / 10 ** (18 - assetDecimal);
    }
```

Listing 2.2: contracts/PumpStaking.sol

Impact The precision loss issue may allow attackers to drain assets or instantly unstake without fees.

Suggestion Properly handle the precision loss.

2.2 Additional Recommendation

2.2.1 Remove redundant code

Status Acknowledged

Introduced by `Version 1`

Description In the `PumpStaking` contract, the `claimAll` function utilizes the `pendingCount` variable to count the user's pending requests and checks that it is a non-zero value (line 251) before transfers. However, this logic is redundant as the check on `totalAmount` already ensures that the user has claimable assets.

```
234    function claimAll() public whenNotPaused {
235        address user = _msgSender();
236        uint256 totalAmount = 0;
237        uint256 pendingCount = 0;
238
239        for(uint8 slot = 0; slot < MAX_DATE_SLOT; slot++) {
240            uint256 amount = pendingUnstakeAmount[user][slot];
241            bool readyToClaim = block.timestamp - pendingUnstakeTime[user][slot] >= (MAX_DATE_SLOT -
                   1) * 1 days;
242            if (amount > 0) {
243                pendingCount += 1;
244                if (readyToClaim) {
245                    totalAmount += amount;
246                    pendingUnstakeAmount[user][slot] = 0;
247                }
248            }
249        }
250
251        require(pendingCount > 0, "PumpBTC: no pending unstake");
252        require(totalAmount > 0, "PumpBTC: haven't reached the claimable time");
253
```

```
    asset.safeTransfer(user, _adjustAmount(totalAmount));

    totalClaimableAmount -= totalAmount;
    totalRequestedAmount -= totalAmount;

    emit ClaimAll(user, totalAmount);
}
```

Listing 2.3: contracts/PumpStaking.sol

Impact N/A

Suggestion Remove the redundant code for gas optimization.

Feedback from the Project The pendingCount is introduced to distinguish between two scenarios where claiming is not allowed:

- There is no pending unstaking.
- Pending requests exist while the claimable time has yet to arrive.

The claimAll function reports different error messages for the above scenarios. This is aligned with the two error reports in the claimSlot function.

2.2.2 Add checks on the new staking limit

Status Fixed in Version 2

Introduced by Version 1

Description In the PumpStaking contract, the setStakeAssetCap function should verify that the newTotalStakingCap parameter is larger than the total staking amount. If totalStakingCap is set to a value lower than the totalStakingAmount, this will cause the stake function to revert, preventing users from staking.

```
function setStakeAssetCap(uint256 newTotalStakingCap) public onlyOwner {
    emit SetStakeAssetCap(totalStakingCap, newTotalStakingCap);
    totalStakingCap = newTotalStakingCap;
}
```

Listing 2.4: contracts/PumpStaking.sol

```
function stake(uint256 amount) public whenNotPaused {
    require(amount > 0, "PumpBTC: amount should be greater than 0");
    require(
        totalStakingAmount + amount <= totalStakingCap,
        "PumpBTC: exceed staking cap"
    );
```

Listing 2.5: contracts/PumpStaking.sol

Impact Users may be unable to stake if totalStakingCap is set to an improper value.

Suggestion Add checks on the newTotalStakingCap parameter.

2.2.3 Follow CEI pattern in the `PumpStaking` contract

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `PumpStaking` contract, several functions do not follow the common CEI (Checks-Effects-Interactions) programming pattern. This pattern dictates that the state variable should be updated before conducting the external call. For example, the `claimSlot` function invokes `asset.safeTransfer` before updating the state variables. It is recommended to follow the CEI pattern to mitigate potential security risks.

```
function claimSlot(uint8 slot) public whenNotPaused {
    address user = _msgSender();
    uint256 amount = pendingUnstakeAmount[user][slot];

    require(amount > 0, "PumpBTC: no pending unstake");
    require(
        block.timestamp - pendingUnstakeTime[user][slot] >= (MAX_DATE_SLOT - 1) * 1 days,
        "PumpBTC: haven't reached the claimable time"
    );

    asset.safeTransfer(user, _adjustAmount(amount));

    pendingUnstakeAmount[user][slot] = 0;
    totalClaimableAmount -= amount;
    totalRequestedAmount -= amount;

    emit ClaimSlot(user, amount, slot);
}
```

Listing 2.6: contracts/PumpStaking.sol

Impact N/A

Suggestion Update the state variables before making any external calls.

2.3 Note

2.3.1 Potential precision loss in the `unstakeInstant` function

Introduced by `Version 1`

Description In the `PumpStaking` contract, a precision loss issue exists in the `unstakeInstant` function.Specifically, in this function, the calculated fee may also be rounded down to zero. This allows users to avoid the fee and unstake instantly from the contract.

```
function unstakeInstant(uint256 amount) public whenNotPaused {
    address user = _msgSender();
    uint256 fee = amount * instantUnstakeFee / 10000;

    require(amount > 0, "PumpBTC: amount should be greater than 0");
    require(amount <= pendingStakeAmount, "PumpBTC: insufficient pending stake amount");

    pumpBTC.burn(user, amount);
    asset.safeTransfer(user, _adjustAmount(amount - fee));

    totalStakingAmount -= amount;
    pendingStakeAmount -= amount;
    collectedFee += fee;

    emit UnstakeInstant(user, amount);
}
```

Listing 2.7: contracts/PumpStaking.sol

Feedback from the Project The fee calculation logic used in the `unstakeInstant` is aligned with the design.

2.3.2 About the off-chain logic

Introduced by `Version 1`

Description The `operator` role of the `PumpStaking` contract has the ability to deposit and withdraw assets (e.g., WBTC) into Babylon to earn rewards. However, the logic for earning and distributing rewards is controlled by off-chain processes, which fall outside the scope of this audit.

Additionally, the `withdrawAndDeposit` function of the `PumpStaking` contract aims to streamline operations by combining withdrawal and deposit processes. However, if the `depositAmount` exceeds both the `pendingStakeAmount` and the required amount, surplus funds intended for staking in Babylon will inadvertently be locked in this contract. While this situation can be managed, it's advisable to carefully conduct off-chain calculations to determine the appropriate `depositAmount` and avoid complicated remedy operations.

```
function withdrawAndDeposit(uint256 depositAmount) public onlyOperator {
    if (pendingStakeAmount > depositAmount) {
        asset.safeTransfer(_msgSender(), _adjustAmount(pendingStakeAmount - depositAmount));
    }
    else if (pendingStakeAmount < depositAmount){
        asset.safeTransferFrom(
            _msgSender(), address(this), _adjustAmount(depositAmount - pendingStakeAmount)
        );
    }

    emit AdminWithdraw(_msgSender(), pendingStakeAmount);
    emit AdminDeposit(_msgSender(), depositAmount);

    pendingStakeAmount = 0;
    totalClaimableAmount += depositAmount;
}
```

Listing 2.8: contracts/PumpStaking.sol

2.3.3 Potential centralization risks

Introduced by Version 1

Description The PumpStaking contract carries potential centralization risks. Specifically, the privileged owner role has the ability to perform sensitive operations, such as pausing/unpausing the contract and modifying key configurations. Additionally, the operator role of this contract is granted the ability to deposit and withdraw assets (e.g., WBTC) into and from the contract. This unavoidably introduces centralization risks, as compromising these key accounts could lead to incorrect functionality of the entire protocol.

In Version 3, the owner is further granted the ability to enable or disable any unstake operations via the setOnlyAllowStake function. The project confirms that unstake operations will be disabled for a few months after Babylon's launch, during which BTC withdrawals from Babylon will be forbidden. Unstaking operations will be allowed once Babylon opens the withdrawal functionality.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS