SMART CONTRACT AUDIT REPORT

for

BTC+

Prepared By: Shuxiao Wang

PeckShield

April 25, 2021

Document Properties

Client	BTC+
Title	Smart Contract Audit Report
Target	BTC+
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	April 25, 2021	Xuxian Jiang	Release Candidate
0.3	April 24, 2021	Xuxian Jiang	Additional Findings #2
0.2	April 14, 2021	Xuxian Jiang	Additional Findings #1
0.1	April 5, 2021	Xuxian Jiang	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **BTC+** protocol, we outline in this report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About BTC+

The `BTC+` protocol is designed to address limitations in current multiple `BTC`-related `ERC20` tokens. Specifically, `BTC`-pegged tokens maintain a stable peg against native `BTC` and token holders must seek for yields across various applications while bearing exorbitant transaction fees associated with allocation adjustments. `BTC LP` tokens include vault share tokens from current farming solutions, e.g., `yEarn`/`Pickle`/`Harvest` and generate profits and socialize costs for their holders. However, these `BTC LP` tokens lose their peg against `BTC` and thus limit their usage in certain applications such as staking. With recurring positive rebase based on accrued interest, `BTC+` aims to both maintain its peg to BTC and provide global interest to all token holders.

The basic information of BTC+ is as follows:

Table 1.1: Basic Information of BTC+

Item	Description
Issuer	BTC+
Website	https://acoconut.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 25, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/nutsfinance/BTC-Plus.git (9ae3dea)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/nutsfinance/BTC-Plus.git (TBD)

1.2 About PeckShield

PeckShield Inc. [15] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [14]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [13], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the BTC+ implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	1	■
High	0	
Medium	3	■ ■ ■
Low	4	■ ■ ■ ■
Informational	1	■
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 critical-severity vulnerability, 3 medium-severity vulnerability, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key BTC+ Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Critical	Arbitrary Share Increment in Base Tokens	Business Logic	Fixed
PVE-002	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-003	Low	Proper harvest() In BadgerRenCRV+	Business Logic	Fixed
PVE-004	Medium	Improper Staking Amount In setRewards()	Business Logic	
PVE-005	Low	Possible Costly Pool LPs From Improper Initialization	Time and State	
PVE-006	Low	Hardcoded Business Logic In redeemBTCBPlus()	Business Logic	
PVE-007	Medium	Trust Issue Of Admin Keys	Security Features	
PVE-008	Informational	Unused Event Removal in GaugeController	Coding Practices	
PVE-009	Medium	Possible Sandwich/MEV For Reduced Gains	Time and State	

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Arbitrary Share Increment in Base Tokens

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Plus
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

BTC+ is an innovative approach that addresses current limitations of BTC-related tokens. Specifically, BTC-pegged tokens maintain a stable peg against native BTC and token holders must seek for yields across various applications while bearing exorbitant transaction fees associated with allocation adjustments. BTC LP tokens include vault share tokens from current farming solutions and generate profits and socialize costs for their holders, but lose their peg against BTC. In the following, we examine the ERC20 implementation of BTC+ that both maintains its peg to BTC and provide global interest to all token holders. The key relies on the unique recurring positive rebasing mechanism from accrued interest.

In the following, we report an issue in the underlying tokenization logic that allows for arbitrary balance increments. Specifically, the base token is inherited from the ERC20Upgradeable contract with an overwritten _transfer() routine to apply necessary rebasing from accrued interest. To elaborate, we show below the _transfer() routine.

```
    /**
     * @dev Moves tokens `amount` from `sender` to `recipient`.
     */
    function _transfer(address _sender, address _recipient, uint256 _amount) internal
        virtual override {
        // Rebase first to make index up-to-date
        rebase();
        uint256 _shareToTransfer = _amount.mul(WAD).div(index);
```

```
218        uint256 _oldSenderShare = userShare[_sender];
219        uint256 _newSenderShare = _oldSenderShare.sub(_shareToTransfer, "insufficient
               share");
220        uint256 _oldRecipientShare = userShare[_recipient];
221        uint256 _newRecipientShare = _oldRecipientShare.add(_shareToTransfer);
222        uint256 _totalShares = totalShares;

224        userShare[_sender] = _newSenderShare;
225        userShare[_recipient] = _newRecipientShare;

227        emit UserShareUpdated(_sender, _oldSenderShare, _newSenderShare, _totalShares);
228        emit UserShareUpdated(_recipient, _oldRecipientShare, _newRecipientShare,
               _totalShares);
229    }
```

Listing 3.1: Plus::_transfer()

It comes to our attention that this routine does not properly handle a corner case when both the sender and the recipient refer to the same account. As a result, the account's balance `userShare[_recipient]` (line 225) can be always incremented without effecting the deduction at line 224. With an increased `userShare`, the malicious actor can drain all funds in the current pools.

Recommendation Revise the `_transfer()` logic to properly handle the corner case when `_sender == _recipient`.

Status

3.2 Improved Precision By Multiplication And Division Reordering

- ID: PVE-009
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `SinglePlus`
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [1]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `SinglePlus::mint()` as an example. This routine is used to mint the single `BTC+` with the underlying token..

```
72     function mint(uint256 _amount) external override nonReentrant {
73         require(_amount > 0, "zero amount");
74         require(!mintPaused, "mint paused");

76         // Rebase first to make index up-to-date
77         rebase();

79         // Transfers the underlying token in.
80         IERC20Upgradeable(token).safeTransferFrom(msg.sender, address(this), _amount);
81         // Conversion rate is the amount of single plus token per underlying token, in
               WAD.
82         uint256 _newAmount = _amount.mul(_conversionRate()).div(WAD);
83         // Index is in WAD
84         uint256 _share = _newAmount.mul(WAD).div(index);

86         uint256 _oldShare = userShare[msg.sender];
87         uint256 _newShare = _oldShare.add(_share);
88         uint256 _totalShares = totalShares.add(_share);
89         totalShares = _totalShares;
90         userShare[msg.sender] = _newShare;

92         emit UserShareUpdated(msg.sender, _oldShare, _newShare, _totalShares);
93         emit Minted(msg.sender, _amount, _share, _newAmount);
94     }
```

Listing 3.2: SinglePlus::mint()

We notice the calculation of `_share` (line 84) involves mixed multiplication and devision. To avoid unnecessary precision loss, it is better to validate with the following requirement: `_share = _amount.mul(_conversionRate()).div(index)`.

It is important to that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in this commit: `8178bf`.

3.3 Proper harvest() In BadgerRenCRV+

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: BadgerRenCRV+
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

The BTC+ protocol is architecturally designed to have a common standard APIs including mint(), redeem(), and harvest(). The mint() operation allows for users to deposit BTC-pegged tokens to get BTC+ tokens, which can then later redeem()'ed. The harvest() operation enables the yield collection. In the following, we examine a specific harvest() implementation of the BadgerRenCRV+ contract.

To elaborate, we show below the harvest() routine. It is designed to firstly harvest from Badger Tree, then convert the collected BADGER/DIGG rewards to WBTC, next deposit WBTC to the Ren Curve pool to get the renCrv share, which is further converted to brenCrv, and finally perform the invest and rebase operations. It comes to our attention that the conversion of DIGG to WBTC is taking an incorrect conversion path, which could revert the harvest() operation. Specifically, the currently used conversion path is DIGG-> WBTC ->address(0) (lines $110 - 112$) and a proper conversion path should be DIGG-> WETH -> WBTC.

```
85  function harvest(address[] calldata _tokens, uint256[] calldata _cumulativeAmounts,
86      uint256 _index, uint256 _cycle, bytes32[] calldata _merkleProof) public virtual
        onlyStrategist {
87      // 1. Harvest from Badger Tree
88      IBadgerTree(BADGER_TREE).claim(_tokens, _cumulativeAmounts, _index, _cycle,
        _merkleProof);
89
90      // 2. Badger --> WETH --> WBTC
91      uint256 _badger = IERC20Upgradeable(BADGER).balanceOf(address(this));
92      if (_badger > 0) {
93          IERC20Upgradeable(BADGER).safeApprove(UNISWAP, 0);
94          IERC20Upgradeable(BADGER).safeApprove(UNISWAP, _badger);
95
96          address[] memory _path = new address[](3);
97          _path[0] = BADGER;
98          _path[1] = WETH;
99          _path[2] = WBTC;
100
101         IUniswapRouter(UNISWAP).swapExactTokensForTokens(_badger, uint256(0), _path,
            address(this), block.timestamp.add(1800));
102     }
103
104     // 3: Digg --> WBTC
105     uint256 _digg = IERC20Upgradeable(DIGG).balanceOf(address(this));
```

```
        if (_digg > 0) {
            IERC20Upgradeable(DIGG).safeApprove(UNISWAP, 0);
            IERC20Upgradeable(DIGG).safeApprove(UNISWAP, _digg);

            address[] memory _path = new address[](3);
            _path[0] = DIGG;
            _path[1] = WBTC;

            IUniswapRouter(UNISWAP).swapExactTokensForTokens(_digg, uint256(0), _path,
                address(this), block.timestamp.add(1800));
        }
        ...
    }
```

Listing 3.3: BadgerRenCRV+::harvest()

Recommendation Correct the conversion path from `DIGG` to `WBTC` in the affected `harvest()`.

Status This issue has been fixed in this commit: `8178bf`.

3.4 Improper Staking Amount In setRewards()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LiquidityGauge`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

The `BTC+` protocol has developed a governance subsystem that can reward participating holders. In particular, the governance subsystem is heavily inspired by the `Curve DAO` implementation and shares the same components, i.e., `GaugeController`, `LiquidityGauge`, and `VotingEscrow`. In the following, we examine a core routine from the `LiquidityGauge` contract.

Specifically, the routine under examination is `setRewards()`, which is designed to update the reward contract and reward tokens. This routine is a permissioned one and can only be invoked by the privileged governance for the reward contract update. Its business logic is implemented as follows: it withdraws all staked assets from the current one and then deposits into the new reward contract. It comes to our attention that the deposit into the new reward has an issue in using an incorrect staking amount. Currently, it calculates the staking amount from `totalSupply()` (line 475), which should be corrected as `IERC20Upgradeable(token).balanceOf(this)`.

```
462    function setRewards(address _rewardContract, address[] memory _rewardTokens)
           external onlyGovernance {
```

```
        address _currentRewardContract = rewardContract;
        address _token = token;
        if (_currentRewardContract != address(0x0)) {
            _checkpointRewards(address(0x0));
            IUniPool(_currentRewardContract).exit();

            IERC20Upgradeable(_token).safeApprove(_currentRewardContract, 0);
        }

        if (_rewardContract != address(0x0)) {
            require(_rewardTokens.length > 0, "reward tokens not set");
            IERC20Upgradeable(_token).safeApprove(_rewardContract, uint256(int256(-1)));
            IUniPool(_rewardContract).stake(totalSupply());

            rewardContract = _rewardContract;
            rewardTokens = _rewardTokens;

            // Complete an initial checkpoint to make sure that everything works.
            _checkpointRewards(address(0x0));

            // Reward contract is tokenized as well
            unsalvageable[_rewardContract] = true;
            // Don't salvage any reward token
            for (uint256 i = 0; i < _rewardTokens.length; i++) {
                unsalvageable[_rewardTokens[i]] = true;
            }
        }

        emit RewardContractUpdated(_currentRewardContract, _rewardContract,
            _rewardTokens);
    }
```

Listing 3.4: LiquidityGauge::setRewards()

Recommendation Correct the `setRewards()` logic by calculating the right staking amount.

Status

3.5 Possible Costly Pool LPs From Improper Initialization

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: LiquidityGauge
- Category: Time and State [8]
- CWE subcategory: CWE-362 [3]

Description

As mentioned in Section 3.4, the BTC+ protocol has developed a governance subsystem that can reward participating holders. And its implementation shares the following main components, i.e., GaugeController, LiquidityGauge, and VotingEscrow. The LiquidityGauge contract allows users to deposit the supported token and get in return gauge-associated pool tokens to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the deposit() routine. This routine is used for participating users to deposit the supported asset and get respective pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function deposit(uint256 _amount) external nonReentrant {
        require(_amount > 0, "zero amount");
        if (_amount == uint256(int256(-1))) {
            // -1 means deposit all
            _amount = IERC20Upgradeable(token).balanceOf(msg.sender);
        }

        _checkpoint(msg.sender);
        _checkpointRewards(msg.sender);

        uint256 _totalSupply = totalSupply();
        uint256 _balance = IERC20Upgradeable(token).balanceOf(address(this));
        // Note: Ideally, when _totalSupply = 0, _balance = 0.
        // However, it's possible that _balance != 0 when _totalSupply = 0, e.g.
        // 1) There are some leftover due to rounding error after all people withdraws;
        // 2) Someone sends token to the liquidity gauge before there is any deposit.
        // Therefore, when either _totalSupply or _balance is 0, we treat the gauge is
            empty.
        uint256 _mintAmount = _totalSupply == 0 || _balance == 0 ?
                _amount : _amount.mul(_totalSupply).div(_balance);

        _mint(msg.sender, _mintAmount);
        _updateLiquidityLimit(msg.sender);

        IERC20Upgradeable(token).safeTransferFrom(msg.sender, address(this), _amount);
```

```
370
371         address _rewardContract = rewardContract;
372         if (_rewardContract != address(0x0)) {
373             IUniPool(_rewardContract).stake(_amount);
374         }
375
376         emit Deposited(msg.sender, _amount, _mintAmount);
377     }
```

Listing 3.5: LiquidityGauge::deposit()

Specifically, when the pool is being initialized (lines 363 – 364), the share amount directly takes the value of `_amount` (line 364), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `_mintAmount = _amount = 1 WEI`. With that, the actor can further deposit a huge amount of token with the goal of making the pool token extremely expensive.

An extremely expensive pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of calculating the `_mintAmount` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status

3.6 Hardcoded Business Logic In redeemBTCBPlus()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BTCZapBsc
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

The BTC+ protocol also provides a number of convenience contracts to facilitate the protocol adoption. In the following, we examine one specific convenience contract BTCZapBsc. As the name indicates, this contract is developed to facilitate the mint() and redeem() opertions of BTC+ tokens.

Specifically, we show below the redeemBTCBPlus() routine that aims to redeems BTCB+ back to BTCB. It firstly converts the composite BTCB+ token back to the single BTCB+ in the form of vBTC+/acsBTCB+, which is further redeemed into vBTC/acsBTCB, and finally to BTCB. It comes to our attention that the current conversion path includes the hardcoded acsBTCB+, which may limit the generic protocol design.

```
    function redeemBTCBPlus(uint256 _amount) public {
        require(_amount > 0, "zero amount");

        IERC20Upgradeable(BTCB_PLUS).safeTransferFrom(msg.sender, address(this), _amount);
        ICompositePlus(BTCB_PLUS).redeem(_amount);

        uint256 _vbtcPlus = IERC20Upgradeable(VENUS_BTC_PLUS).balanceOf(address(this));
        ISinglePlus(VENUS_BTC_PLUS).redeem(_vbtcPlus);
        uint256 _vbtc = IERC20Upgradeable(VENUS_BTC).balanceOf(address(this));
        IVToken(VENUS_BTC).redeem(_vbtc);

        uint256 _acsBtcbPlus = IERC20Upgradeable(ACS_BTCB_PLUS).balanceOf(address(this));
        ISinglePlus(ACS_BTCB_PLUS).redeem(_acsBtcbPlus);
        uint256 _acsBtcb = IERC20Upgradeable(ACS_BTCB).balanceOf(address(this));
        IVault(ACS_BTCB).withdraw(_acsBtcb);

        uint256 _btcb = IERC20Upgradeable(BTCB).balanceOf(address(this));
        IERC20Upgradeable(BTCB).safeTransfer(msg.sender, _btcb);

        emit Redeemed(msg.sender, BTCB_PLUS, _btcb, _amount);

    }
```

Listing 3.6: BTCZapBsc::redeemBTCBPlus()

Recommendation Avoid hard-coding a specific intermediate token in the `redeemBTCBPlus()` logic.

Status This issue has been confirmed. The team has informed us the current protocol only supports `vBTC+` and `acsBTCB+` in `BTCB+`. With new components are supported, the convenience contract will be upgraded with the new support.

3.7 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In the `BTC+` protocol, there is a special administrative account, i.e., `governance`. This `governance` account plays a critical role in governing and regulating the system-wide operations (e.g., account whitelisting and risk parameter setting). It also has the privilege to regulate or govern the flow of assets among the involved components.

With great privilege comes great responsibility. Our analysis shows that the `governance` account is indeed privileged. In the following, we show representative privileged operations in the `BTC+` protocol.

```
    /**
     * @dev Updates governance. Only governance can update governance.
     */
    function setGovernance(address _governance) external onlyGovernance {
        address _oldGovernance = governance;
        governance = _governance;
        emit GovernanceUpdated(_oldGovernance, _governance);
    }

    /**
     * @dev Updates claimer. Only governance can update claimers.
     */
    function setClaimer(address _account, bool _allowed) external onlyGovernance {
        claimers[_account] = _allowed;
        emit ClaimerUpdated(_account, _allowed);
    }

    /**
     * @dev Updates the AC emission base rate for plus gauges. Only governance can
       update the base rate.
     */
```

```
    function setPlusReward(uint256 _plusRewardPerDay) external onlyGovernance {
        uint256 _oldRate = basePlusRate;
        // Base rate is in WAD
        basePlusRate = _plusRewardPerDay.mul(WAD).div(DAY);
        // Need to checkpoint with the base rate update!
        checkpoint();

        emit BasePlusRateUpdated(_oldRate, basePlusRate);
    }
```

Listing 3.7: Various Setters in GaugeController

We emphasize that the privilege assignment with various core contracts is necessary and required for proper protocol operations. However, it is worrisome if the `governance` account is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account.

We point out that a compromised `governance` account would allow the attacker to maliciously change protocol settings to compromise funds, which directly undermines the assumption of the `BTC+` protocol. Also, we point out that the current contract deployment makes use of the proxy-based approach, which emphasizes the similar admin key issue of the privileged proxy-admin account.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status

3.8 Unused Event Removal in GaugeController

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `GaugeController`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [4]

Description

The `BTC+` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Initializable`, to facilitate its code implementation and organization. For example, the `GaugeController` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `FeeProcessed` event (line 40), this event has been defined in `GaugeController`, but never used or emitted. With that, this event can be safely removed.

```
31    event GovernanceUpdated(address indexed oldGovernance, address indexed newGovernance
          );
32    event ClaimerUpdated(address indexed claimer, bool allowed);
33    event BasePlusRateUpdated(uint256 oldBaseRate, uint256 newBaseRate);
34    event TreasuryUpdated(address indexed oldTreasury, address indexed newTreasury);
35    event GaugeAdded(address indexed gauge, bool plus, uint256 gaugeWeight, uint256
          gaugeRate);
36    event GaugeRemoved(address indexed gauge);
37    event GaugeUpdated(address indexed gauge, uint256 oldWeight, uint256 newWeight,
          uint256 oldGaugeRate, uint256 newGaugeRate);
38    event Checkpointed(uint256 oldRate, uint256 newRate, uint256 totalSupply, uint256
          ratePerToken, address[] gauges, uint256[] guageRates);
39    event RewardClaimed(address indexed gauge, address indexed user, address indexed
          receiver, uint256 amount);
40    event FeeProcessed(address indexed gauge, address indexed token, uint256 amount);
```

Listing 3.8: Various events defined in GaugeController

Recommendation Consider the removal of the unused event `FeeProcessed`.

Status

3.9 Possible Sandwich/MEV For Reduced Gains

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

As mentioned in Section 3.3, the `BTC+` protocol is architecturally designed to have a common standard APIs including `mint()`, `redeem()`, as well as `harvest()`. In the following, we examine the logic behind `harvest()` operation for yield collection.

To elaborate, we show below the `harvest()` function in the `VenusBTC+` contract. It basically harvests additional yield from the investment by converting rewards back to `vBTC+`.

```
    function harvest() public virtual override onlyStrategist {
        // Harvest from Venus comptroller
        IVenusComptroller(VENUS_COMPTROLLER).claimVenus(address(this));

        // Harvest from VAI controller
        IVAIVault(VAI_VAULT).claim();
```

```

        uint256 _venus = IERC20Upgradeable(VENUS).balanceOf(address(this));
        // PancakeSawp: XVS --> WBNB --> BTCB
        if (_venus > 0) {
            IERC20Upgradeable(VENUS).safeApprove(PANCAKE_SWAP_ROUTER, 0);
            IERC20Upgradeable(VENUS).safeApprove(PANCAKE_SWAP_ROUTER, _venus);

            address[] memory _path = new address[](3);
            _path[0] = VENUS;
            _path[1] = WBNB;
            _path[2] = BTCB;

            IUniswapRouter(PANCAKE_SWAP_ROUTER).swapExactTokensForTokens(_venus, uint256
                (0), _path, address(this), block.timestamp.add(1800));
        }
        // Venus: BTCB --> vBTC
        uint256 _btcb = IERC20Upgradeable(BTCB).balanceOf(address(this));
        if (_btcb == 0) return;

        // If there is performance fee, charged in BTCB
        uint256 _fee = 0;
        if (performanceFee > 0) {
            _fee = _btcb.mul(performanceFee).div(PERCENT_MAX);
            IERC20Upgradeable(BTCB).safeTransfer(treasury, _fee);
            _btcb = _btcb.sub(_fee);
        }

        IERC20Upgradeable(BTCB).safeApprove(VENUS_BTC, 0);
        IERC20Upgradeable(BTCB).safeApprove(VENUS_BTC, _btcb);
        IVToken(VENUS_BTC).mint(_btcb);

        // Reinvest to get compound yield.
        _invest();
        // Also it's a good time to rebase!
        rebase();

        emit Harvested(VENUS_BTC, _btcb, _fee);
    }
```

Listing 3.9: VenusBTC+::harvest()

We notice the above conversion leverages `PancakeSwap` in order to swap related rewards to `BTCB`. And the swap operation does not specify a valid restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller converted amount. Note other contracts, e.g., `AcryptoSBTC+`, `AutoBTC+/AutoBTCv2+`, and `ForTubeBTCB+`, share the same issue.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the virtual account in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of protocol users. Note that the current authenticated call to `harvest()` with the `onlyStrategist` modifier mitigates this issue.

Status

4 Conclusion

In this audit, we have analyzed the BTC+ design and implementation. The system presents a unique offering as a recurring positive rebasing mechanism that can not only maintain its peg to BTC, but also provide global interest to all token holders. The current code base is well organized and those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[7] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[8] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[9] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[10] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[11] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[12] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[13] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[14] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[15] PeckShield. PeckShield Inc. https://www.peckshield.com.