BLOCKSEC

# Security Audit

# Report for brBTC & brVault Contracts

**Date:** December 16, 2024 **Version:** 1.0

**Contact:** contact@blocksec.com

# Contents

## Report Manifest

| Item | Description |
| --- | --- |
| Client | Bedrock |
| Target | brBTC & brVault Contracts |

## Version History

| Version | Date | Description |
| --- | --- | --- |
| 1.0 | December 16, 2024 | First release |

## Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-12-16 09:11:29+0000

# Chapter 1 Introduction

## 1.1 About Target Contracts

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

This audit focuses on the code repositories of Bedrock's brBTC & brVault Contracts [1]. These contracts enable users to convert their wrapped BTC tokens into `brBTC` and securely store them in the brVault. Notably, the `brVault` contract includes a whitelist mechanism for wrapped BTC tokens to enhance security.

Please note that this audit covers only the following contracts:

- contracts/brBTC.sol
- contracts/brVault.sol

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 |
|---|---|---|
| brBTC & brVault Contracts | `Version 1` | `56c251e9a87b30366cd840baed8330e83c9e6853` |
| | `Version 2` | `f07d71d07bec5115dabf413c8426f90fecd02597` |

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

[1] https://github.com/Bedrock-Technology/omni

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 and Common Weakness Enumeration. 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 \ Likelihood | *High* | *Low* |
|---|---|---|
| *High* | High | Medium |
| *Low* | Medium | Low |

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.
- **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 **one** recommendation and **two** notes.

- Low Risk: 1
- Recommendation: 1
- Note: 2

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | Low | Improper amount truncation | DeFi Security | Fixed |
| 2 | - | Add validation checks in the `setCap` function | Recommendation | Fixed |
| 3 | - | Potential centralization risk | Note | - |
| 4 | - | Lack of entry points for native tokens | Note | - |

The details are provided in the following sections.

## 2.1 DeFi Security

### 2.1.1 Improper amount truncation

**Severity** Low

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the `brVault` contract, the `_mint` function transfers users' wrapped BTC tokens to the vault and mints an equivalent amount of `brBTC` tokens for them at a 1:1 ratio. To accommodate wrapped BTC tokens with varying decimals, the input `_amount` is truncated in the internal `_amounts` function before being used for minting `brBTC` tokens. However, this truncation creates an issue because the `_mint` function still uses the original `_amount` to transfer tokens from users. As a result, when users mint `brBTC` tokens with a wrapped BTC token that has 18 decimals, they receive fewer `brBTC` tokens than the original `_amount` of the wrapped BTC token.

```
279 function _mint(address _sender, address _token, uint256 _amount) internal {
280   uint256 brBTCAmount = _amounts(_token, _amount);
281   require(brBTCAmount > 0, "USR010");
282
283   uint256 tokenUsedCap = tokenUsedCaps[_token];
284   require((tokenUsedCap + _amount < tokenCaps[_token]), "USR003");
285   tokenUsedCaps[_token] = tokenUsedCap + _amount;
286
287   IERC20(_token).safeTransferFrom(_sender, address(this), _amount);
288   IMintableContract(brBTC).mint(_sender, brBTCAmount);
289
290   emit Minted(_token, _amount);
291 }
```

**Listing 2.1:** contracts/brVault.sol

```
298 function _amounts(address _token, uint256 _amount) internal view returns (uint256) {
299   uint8 decs = ERC20(_token).decimals();
300   if (decs == 8) return _amount;
301   if (decs == 18) return _amount / DECIMAL_PRECISION18;
302   return 0;
303 }
```

**Listing 2.2:** contracts/brVault.sol

**Impact** Users may receive fewer `brBTC` tokens due to improper truncation of the input `_amount`.

**Suggestion** Revise the code logic accordingly.

## 2.2 Additional Recommendation

### 2.2.1 Add validation checks in the `setCap` function

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The `setCap` function in the `brVault` contract sets the cap for each type of wrapped BTC. It is recommended to include the following validation checks:

- Verify that the input `_token` is included in the whitelist (`allowedTokens`).
- Ensure that the input `_cap` is greater than or equal to `tokenUsedCap`.

```
165 function setCap(address _token, uint256 _cap) external onlyRole(DEFAULT_ADMIN_ROLE) {
166   require(_token != address(0x0), "SYS003");
167   require(_cap > 0, "USR017");
168
169   uint8 decs = ERC20(_token).decimals();
170
171   require(decs == 8 || decs == 18, "SYS004");
172
173   tokenCaps[_token] = _cap;
174 }
```

**Listing 2.3:** contracts/brVault.sol

**Suggestion** Add validation checks for the inputs `_token` and `_cap` in the `setCap` function.

## 2.3 Note

### 2.3.1 Potential centralization risk

**Introduced by** `Version 1`

**Description** The protocol includes several privileged functions, such as `allowToken`, `denyToken`, and `execute`. If the private key of the admin or operator is compromised or maliciously exploited, it could lead to significant user losses. For example, if the private key associated with the `OPERATOR_ROLE` is compromised, an attacker could use the `execute` function to siphon all funds stored in the vault.

```
236 function execute(address _target, bytes memory _data, uint256 _value)
237   external
238   nonReentrant
239   onlyRole(OPERATOR_ROLE)
240   serviceNormal
241   returns (bytes memory)
242 {
243   require(allowedTargets[_target], "SYS001");
244   return _target.functionCallWithValue(_data, _value);
245 }
```

**Listing 2.4:** contracts/brVault.sol

### 2.3.2 Lack of entry points for native tokens

**Introduced by** `Version 1`

**Description** In the `brVault` contract, the `execute` function allows the operator to execute arbitrary transactions involving native tokens. However, the `brVault` contract does not include any payable functions or implement the `receive` or `fallback` functions to accept native tokens.

```
236   function execute(address _target, bytes memory _data, uint256 _value)
237       external
238       nonReentrant
239       onlyRole(OPERATOR_ROLE)
240       serviceNormal
241       returns (bytes memory)
242   {
243       require(allowedTargets[_target], "SYS001");
244       return _target.functionCallWithValue(_data, _value);
245   }
```

**Listing 2.5:** contracts/brVault.sol


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS