BLOCKSEC

# Security Audit
# Report for UniBTC

**Date:** July 12, 2024 **Version:** 1.0

**Contact:** contact@blocksec.com

# Contents

## Report Manifest

| Item | Description |
| --- | --- |
| Client | Bedrock Technology |
| Target | UniBTC |

## Version History

| Version | Date | Description |
| --- | --- | --- |
| 1.0 | July 12, 2024 | First release |

## Signature

**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 the UniBTC contracts of Bedrock Technology [1]. Users can convert their BTC assets to corresponding amount of `uniBTC` tokens and bridge `uniBTC` tokens among different chains through Celer Network. Please notice that only the three source files (i.e., `uniBTC.sol`, `Peer.sol` and `Vault.sol`) are within the scope of our audit.

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 |
|---|---|---|
| UniBTC | `Version 1` | `30cae66619931c1e5c36c5f4fd8c0f85bd0180f8` |
| | `Version 2` | `TODO` |

## 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.

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] https://github.com/Bedrock-Technology/uniBTC

## 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 \ 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.

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

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

# 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 | Precision loss in minting progress | Software Security | Undetermined |
| 2 | - | Remove unused functionalities | Recommendation | Undetermined |
| 3 | - | Potential centralization risks | Note | - |
| 4 | - | Unpegged tokens may drain the vault | Note | - |

The details are provided in the following sections.

## 2.1 Software Security

### 2.1.1 Precision loss in minting progress

**Severity** Low

**Status** Undetermined

**Introduced by** `Version 1`

**Description** The `Vault` contract accepts several whitelisted tokens to be deposited and mints corresponding amount of `uniBTC` tokens to depositors. The `_amounts` function returns two integers representing the actual token amount to be consumed and the amount of `uniBTC` minted, according to the amount of the deposited tokens.

```
180   function _mint(address _sender, address _token, uint256 _amount) internal {
181       (, uint256 uniBTCAmount) = _amounts(_token, _amount);
182       require(uniBTCAmount > 0, "USR010");
183
184       require(IERC20(_token).balanceOf(address(this)) + _amount <= caps[_token], "USR003");
185
186       IERC20(_token).safeTransferFrom(_sender, address(this), _amount);
187       IMintableContract(uniBTC).mint(_sender, uniBTCAmount);
188
189       emit Minted(_token, _amount);
190   }
```

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

```
229   function _amounts(address _token, uint256 _amount) internal returns (uint256, uint256) {
230       uint8 decs = ERC20(_token).decimals();
231       if (decs == 8) return (_amount, _amount);
232       if (decs == 18) {
233           uint256 uniBTCAmt = _amount /EXCHANGE_RATE_BASE;
```

```
234             return (uniBTCAmt * EXCHANGE_RATE_BASE, uniBTCAmt);
235         }
236         return (0, 0);
237     }
```

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

As shown in the code segments above, in the `_mint` function, the token amount used in the `safeTransferFrom` should be the first return value of `_amounts`. However, the minting progress uses the `_amount` parameter, which may bring precision losses.

Specifically, if the deposited amount of tokens contains dusts (i.e., less than `EXCHANGE_RATE_BASE`), the dust part is trucated, resulting in less `uniBTC` tokens minted.

The problem also exists in the overloaded function to mint `uniBTC` tokens from native tokens (`BTC` itself is the native token in some Bitcoin-based Layer 2 Networks).

```
166     function _mint(address _sender, uint256 _amount) internal {
167         (, uint256 uniBTCAmount) = _amounts(_amount);
168         require(uniBTCAmount > 0, "USR010");
169
170         require(address(this).balance <= caps[NATIVE_BTC], "USR003");
171
172         IMintableContract(uniBTC).mint(_sender, uniBTCAmount);
173
174         emit Minted(NATIVE_BTC, _amount);
175     }
```

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

This function should refund the remaining dusts to the users.

**Impact** Users may spend more tokens to mint the same amount of `uniBTC` tokens.

**Suggestion** Use the correct amount instead.

## 2.2 Additional Recommendation

### 2.2.1 Remove unused functionalities

**Status** Undetermined

**Introduced by** `Version 1`

**Description** The `Vault` contract inherits `PausableUpgradeable` while implements pausing mechanism. However, the `mint` functions implements its own pausing mechanism in the token level, and the inheritance of the `PausableUpgradeable` contract can be removed to save gas.

```
57     function mint(address _token, uint256 _amount) external {
58         require(!paused[_token], "SYS002");
59         _mint(msg.sender, _token, _amount);
60     }
```

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

**Suggestion** Remove unused functionalities.

## 2.3 Note

### 2.3.1 Potential centralization risks

**Introduced by** `Version 1`

**Description** In the UniBTC contracts, there are multiple functions with privileges to set key parameters. Modifying these parameters can significantly alter the functionality of the contracts, potentially rendering them unusable or in an incorrect state.

### 2.3.2 Unpegged tokens may drain the vault

**Introduced by** `Version 1`

**Description** In the `Vault` contract, users can convert any supported tokens (e.g., `WBTC` and `FBTC`) into `uniBTC` tokens and redeem them back with exact the same amount. In other words, users can "swap" the supported tokens at a 1:1 ratio through the `Vault` contract. If one of the supported tokens is unpegged with the underlying BTC, the unpegged token can still be swapped into other tokens at a 1:1 ratio, which may drain the `Vault` and lead to other unexpected consequences.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS