SMART CONTRACT AUDIT REPORT

for

uniBTC

Prepared By: Xiaomi Huang

PeckShield

October 1, 2024

Document Properties

Client	BedRock
Title	Smart Contract Audit Report
Target	uniBTC
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 1, 2024	Xuxian Jiang	Final Release
1.0-rc	September 29, 2024	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the uniBTC protocol, we outline in the 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 contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About uniBTC

uniBTC is a synthetic asset that represents Bitcoin (BTC) on various decentralized finance (DeFi) protocols. Essentially, it is a tokenized version of BTC that allows holders to utilize the value of Bitcoin within the DeFi space, bypassing the limitations of Bitcoin's native blockchain. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Audited Contracts

Item	Description
Target	uniBTC
Website	https://bedrock.technology/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 1, 2024

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

- https://github.com/Bedrock-Technology/uniBTC.git (2851378)

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

- https://github.com/Bedrock-Technology/uniBTC.git (65199be)

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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

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

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) [7], 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 1.4: 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 Logics	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 implementation of the `uniBTC` protocol. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	2	
Undetermined	1	
Total	4	

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 that 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 medium-severity vulnerability, 2 low-severity vulnerability, and and 1 undetermined issue.

Table 2.1: Key uniBTC Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Undetermined	Revisited Default Policy For Token Support in Vault	Coding Practices	Resolved
PVE-002	Low	Improved Token Cap Enforcement in Vault::_mint()	Coding Practices	Resolved
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated
PVE-004	Low	Improved _getDebtTokenAmount() Logic in DelayRedeemRouter	Business Logic	Resolved

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 Revisited Default Policy For Token Support in Vault

- ID: PVE-001
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: Vault
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The uniBTC protocol has a core Vault contract that holds user funds in exchange for uniBTC tokens. The contract is designed to support multiple input tokens that may be used to mint uniBTC tokens. While reviewing the related logic, we notice it adopts a default policy to permit all input tokens and relies on the token-specific cap threshold to enforce whether a specific mint request is allowed. This default approach is strongly suggested to take a token-whitelisting approach to ensure only a whitelisted token will be supported.

In the following, we show the implementation of the related mint() routine. It does check the token-specific paused state and only allows an unpaused token to proceed. This could be problematic as the default state of any token is not paused.

```
    function mint() external payable {
        require(!paused[NATIVE_BTC], "SYS002");
        _mint(msg.sender, msg.value);
    }

    /**
     * @dev mint uniBTC with the given type of wrapped BTC
     */
    function mint(address _token, uint256 _amount) external {
        require(!paused[_token], "SYS002");
        _mint(msg.sender, _token, _amount);
    }
```

Listing 3.1: Vault::mint()

Similarly, the `Vault` contract has a privileged routine `execute()` to execute a low-level contract call. This low-level contract call is very powerful and can be used to transfer funds out of the vault. With that, we strongly suggest to take a whitelist approach as well to ensure only a whitelisted target may be invoked.

```
function execute(address _target, bytes memory _data, uint256 _value) external
    nonReentrant onlyRole(OPERATOR_ROLE) returns(bytes memory) {

    return _target.functionCallWithValue(_data, _value);
}
```

Listing 3.2: `Vault::execute()`

Recommendation Improve the above-mentioned routines to ensure only intended tokens (or targets) are allowed to proceed.

Status This issue has been fixed in the following commit: `fdfedb9`.

3.2 Improved Token Cap Enforcement in Vault::_mint()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `uniBTC` protocol is no exception. Specifically, if we examine the `Vault` contract, it has defined a number of protocol-wide risk parameters, such as `paused` and `caps` for each specific token. In the following, we show the corresponding routines that enforce the token's cap limit when minting `uniBTC`.

```
function _mint(address _sender, uint256 _amount) internal {
    (, uint256 uniBTCAmount) = _amounts(_amount);
    require(uniBTCAmount > 0, "USR010");

    uint256 totalSupply = ISupplyFeeder(supplyFeeder).totalSupply(NATIVE_BTC);
    require(totalSupply <= caps[NATIVE_BTC], "USR003");

    IMintableContract(uniBTC).mint(_sender, uniBTCAmount);

    emit Minted(NATIVE_BTC, _amount);
}
```

```

    /**
     * @dev mint uniBTC with wrapped BTC tokens
     */
    function _mint(address _sender, address _token, uint256 _amount) internal {
        (, uint256 uniBTCAmount) = _amounts(_token, _amount);
        require(uniBTCAmount > 0, "USR010");

        uint256 totalSupply = ISupplyFeeder(supplyFeeder).totalSupply(_token);
        require(totalSupply + _amount <= caps[_token], "USR003");

        IERC20(_token).safeTransferFrom(_sender, address(this), _amount);
        IMintableContract(uniBTC).mint(_sender, uniBTCAmount);

        emit Minted(_token, _amount);
    }
```

Listing 3.3: `Vault::_mint()`

The above enforcement can be improved to explicitly ensure the token-specific caps limit is not zero. By doing so, we can guarantee the native coin minting issue does not issue if its caps limit is set to zero. This also serves as the intended second checkpoint for threshold enforcement in addition to the token whitelisting requirement.

Recommendation Improve the above routine to ensure the token-specific threshold is non-zero.

Status This issue has been fixed in the following commit: `fdfedb9`.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `uniBTC` protocol contract, there is a privileged account (with the assigned `DEFAULT_ADMIN_ROLE` role) that plays a critical role in governing and regulating the DAO-wide operations (e.g., assign roles, configure parameters, and upgrade proxies). In the following, we show the representative functions potentially affected by the privilege of this account.

```
    function pauseToken(address _token) public onlyRole(PAUSER_ROLE) {
        paused[_token] = true;
        emit TokenPaused(_token);
    }
```

```
    /**
     * @dev a pauser unpause the minting of a token
     */
    function unpauseToken(address _token) public onlyRole(PAUSER_ROLE) {
        paused[_token] = false;
        emit TokenUnpaused(_token);
    }


    /**
     * @dev set cap for a specific type of wrapped BTC
     */
    function setCap(address _token, uint256 _cap) external onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_token != address(0x0), "SYS003");

        uint8 decs = NATIVE_BTC_DECIMALS;

        if (_token != NATIVE_BTC) decs = ERC20(_token).decimals();

        require(decs == 8  decs == 18, "SYS004");

        caps[_token] = _cap;
    }

    /**
     * @dev set the supply feeder address to track the asset supply for the vault
     */
    function setSupplyFeeder(address _supplyFeeder) external onlyRole(DEFAULT_ADMIN_ROLE) {
        supplyFeeder = _supplyFeeder;
    }
```

Listing 3.4: Example Privileged Operations in `Vault`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive vault parameters, which directly undermines the assumption of the vault design.

In the meantime, the vault contract and others make use of the proxy contract to allow for future upgrades. Their upgrade is a privileged operation, which also falls in this trust issue on the admin key.

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 The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the privileged account.

3.4 Improved _getDebtTokenAmount() Logic in DelayRedeemRouter

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Medium
- Target: `DelayRedeemRouter`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `uniBTC` protocol has the built-in `DelayRedeemRouter` contract to handle user redemption requests. Accordingly, it provides a key helper routine to get the claimable debt list. In the process of examining the helper routine to calculate the claimable debt list, we notice current implementation can be improved.

To elaborate, we show below the implementation of the related `_getDebtTokenAmount()` routine. As the name indicates, this routine is used to obtain the claimable debt list. It comes to our attention that the calculated list may contain redundant entries at the end of list especially when the debt tokens are being merged (line 773). These redundant entries are suggested for removal.

```
732     function _getDebtTokenAmount(
733         address recipient,
734         uint256 delayedRedeemsCompletedBefore,
735         uint256 delayTimestamp,
736         uint256 maxNumberOfDelayedRedeemsToClaim
737     ) internal view returns (uint256, DebtTokenAmount[] memory) {
738         uint256 _userRedeemsLength = _userRedeems[recipient]
739             .delayedRedeems
740             .length;
741         uint256 numToClaim = 0;
742         while (
743             numToClaim < maxNumberOfDelayedRedeemsToClaim &&
744             (delayedRedeemsCompletedBefore + numToClaim) < _userRedeemsLength
745         ) {
746             // copy delayedRedeem from storage to memory
747             DelayedRedeem memory delayedRedeem = _userRedeems[recipient]
748                 .delayedRedeems[delayedRedeemsCompletedBefore + numToClaim];
749             // check if delayedRedeem can be claimed. break the loop as soon as a
                    delayedRedeem cannot be claimed
750             if (
751                 block.timestamp <
752                 delayedRedeem.timestampCreated + delayTimestamp
```

```
            ) {
                break;
            }
            // increment i to account for the delayedRedeem being claimed
            unchecked {
                ++numToClaim;
            }
        }

        if (numToClaim > 0) {
            DebtTokenAmount[] memory debtAmounts = new DebtTokenAmount[](
                numToClaim
            );
            uint256 tempCount = 0;
            for (uint256 i = 0; i < numToClaim; i++) {
                DelayedRedeem memory delayedRedeem = _userRedeems[recipient]
                    .delayedRedeems[delayedRedeemsCompletedBefore + i];
                bool found = false;

                for (uint256 j = 0; j < tempCount; j++) {
                    if (debtAmounts[j].token == delayedRedeem.token) {
                        debtAmounts[j].amount += delayedRedeem.amount;
                        found = true;
                        break;
                    }
                }
                if (!found) {
                    debtAmounts[tempCount] = DebtTokenAmount({
                        token: delayedRedeem.token,
                        amount: delayedRedeem.amount
                    });
                    tempCount++;
                }
            }
            return (numToClaim, debtAmounts);
        }

        return (0, new DebtTokenAmount[](0));
    }
```

Listing 3.5: `DelayRedeemRouter::_getDebtTokenAmount()`

Recommendation Strengthen the above-mentioned routine to properly remove the redundant entries, if any.

Status This issue has been fixed in the following commit: `61160cb`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `uniBTC` protocol, which is a synthetic asset that represents `Bitcoin (BTC)` on various decentralized finance (`DeFi`) protocols. Essentially, it is a tokenized version of `BTC` that allows holders to utilize the value of `Bitcoin` within the `DeFi` space, bypassing the limitations of `Bitcoin`'s native blockchain. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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