Security Audit Report for UniBTC

Date: November 15, 2024 **Version:** 1.1

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Bedrock Technology
Target	UniBTC

Version History

Version	Date	Description
1.0	October 30, 2024	First release
1.1	November 15, 2024	Update redeem logic

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-11-15 08:21:15+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 the UniBTC contracts of Bedrock Technology [1]. Users can convert their `BTC` assets to an equivalent amount of `uniBTC` tokens and bridge these tokens across different chains through the Chainlink CCIP. Please note that only the source code files in two folders (i.e., `contracts/contracts/` and `ccip/`) are within the scope of this 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`	`b09e38843605bd5cd00d86a970a5ae6f071b2f01`
	`Version 2`	`4b8f09f685f17fd6df08fc46448b3f5df2e74f9c`
	`Version 3`	`ec3a5910a47799ade601d3e44fd6e75190f69108` [2]

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

[2] The audit report was updated to Version 1.1 to reflect the new commit (`Version 3`); we reviewed this commit and found no security issues.

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 [3] and Common Weakness Enumeration [4]. 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.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
[4] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **five** potential security issues. Besides, we have **six** recommendations and **three** notes.

- High Risk: 1
- Low Risk: 4
- Recommendation: 6
- Note: 3

ID	Severity	Description	Category	Status
1	Low	Missing nonce for signature generation	Software Security	Fixed
2	Low	Potential transfer failures due to the use of native `transfer` method	Software Security	Fixed
3	Low	Lack of a refund mechanism for excessive fees	Software Security	Fixed
4	High	Incorrect address for cross-chain message	Software Security	Fixed
5	Low	Potential read-only reentrancy risk	DeFi Security	Fixed
6	-	Remove unnecessary contract inheritance	Recommendation	Fixed
7	-	Add sanity checks when setting crucial variables	Recommendation	Fixed
8	-	Ajust the order of checks to optimize gas usage	Recommendation	Fixed
9	-	Read `mBTC` address from the `mTokenSwap`	Recommendation	Fixed
10	-	Emit events when changing crucial parameters	Recommendation	Fixed
11	-	Remove redundant checks for parameters	Recommendation	Fixed
12	-	Potential centralization risks	Note	-
13	-	Unpegged tokens may drain the vault	Note	-
14	-	The slippage check is applicable only to BTC-pegged tokens	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Missing nonce for signature generation

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Users can bridge tokens by providing a signed message to the `ccipPeer` contract. However, the signature content lacks a nonce field. As a result, the contract cannot distinguish whether a message with identical content is due to normal cross-chain behavior of the `uniBTC` tokens or unexpected issues, such as multiple signatures for the same content caused by client-side or network errors.

```
function verifySendTokenSign(
    address _sender,
    uint64 _destinationChainSelector,
    address _recipient,
    uint256 _amount,
    bytes memory _signature
) public view returns (bool) {
    bytes32 msgDigest =
        sha256(abi.encode(_sender, address(this), block.chainid, _destinationChainSelector,
             _recipient, _amount));
    address signer = ECDSA.recover(msgDigest, _signature);
    return signer == sysSigner;
}
```

Listing 2.1: ccip/src/ccipPeer.sol

```
function sendToken(uint64 _destinationChainSelector, address _recipient, uint256 _amount, bytes
     memory _signature)
    external
    payable
    whenNotPaused
    validateReceiver(_recipient)
    returns (bytes32 messageId)
{
    require(_amount >= minTransferAmt, "USR006");
    require(
        verifySendTokenSign(msg.sender, _destinationChainSelector, _recipient, _amount,
             _signature), "SIGNERROR"
    );
    if (processedSignature[_signature]) revert SignatureProcessed();
    processedSignature[_signature] = true;
    return _sendToken(_destinationChainSelector, _recipient, _amount);
}
```

Listing 2.2: ccip/src/ccipPeer.sol

Impact The `ccipPeer` contract may subject to unexpected behaviors.

Suggestion Add a nonce field in the signature content.

2.1.2 Potential transfer failures due to the use of native `transfer` method

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `DelayRedeemRouter` contract, the native `transfer` method is used to transfer native tokens. However, this transfer can fail when the recipients are contracts. Specifically, the `transfer` method limits the gas usage of the underlying `call`. If the recipient is a contract that implements logic in its `receive` function, the transfer may fail due to insufficient gas.

```
if (token == NATIVE_BTC) {
    // transfer native token to the recipient
    IVault(vault).execute(address(this), "", amountToSend);
    payable(recipient).transfer(amountToSend);
```

Listing 2.3: contracts/contracts/proxies/stateful/redeem/DelayRedeemRouter.sol

```
function _approveUnbound(uint256 reqId) internal {
    uint256 amount = withdrawPendingQueue[reqId];
    // not found, do nothing.
    if (amount > 0) {
        withdrawPendingAmount -= amount;
        delete withdrawPendingQueue[reqId];
        payable(vault).transfer(amount);
    }
    emit UnboundApproved(reqId, "");
}
```

Listing 2.4: ccontracts/proxies/stateful/BitLayerNativeProxy.sol

Impact Native token transfers may fail due to the use of native `transfer`.

Suggestion Use a low-level `call` to prevent potential failures in native token transfers.

2.1.3 Lack of a refund mechanism for excessive fees

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `targetCall` function receives the `msg.value` and invokes the router with a `fees` value. If the `msg.value` exceeds the required `fees`, the excess amount is not refunded and remains in the contract.

```
function targetCall(uint64 _destinationChainSelector, address _target, bytes memory _callData)
    external
    payable
    whenNotPaused
    onlyRole(DEFAULT_ADMIN_ROLE)
    returns (bytes32 messageId)
{
    address _receiver = allowlistedDestinationChains[_destinationChainSelector];
    if (_receiver == address(0)) {
        revert DestinationChainNotAllowlisted(_destinationChainSelector, _receiver);
    }
    bytes memory _message = abi.encode(Request({target: _target, callData: _callData}));
    Client.EVM2AnyMessage memory evm2AnyMessage = _buildCCIPMessage(_receiver, _message,
        address(0));
    // Initialize a router client instance to interact with cross-chain router
    IRouterClient router = IRouterClient(this.getRouter());
    // Get the fee required to send the CCIP message
    uint256 fees = router.getFee(_destinationChainSelector, evm2AnyMessage);
```

```
        require(msg.value >= fees, "USR008");
        // Send the CCIP message through the router and store the returned CCIP message ID
        messageId = router.ccipSend{value: fees}(_destinationChainSelector, evm2AnyMessage);
        // Emit an event with message details
        emit MessageSent(messageId, _destinationChainSelector, _receiver, _message, address(0),
            fees);
        return messageId;
    }
```

Listing 2.5: ccip/src/ccipPeer.sol

Impact Excessive fees are not refunded to the sender.

Suggestion Add a refunding mechanism for excessive fees.

2.1.4 Incorrect address for cross-chain message

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `estimateSendTokenFees` function of the `ccipPeer` contract, the `_target` address (i.e., the `uniBTC` token address on the destination chain) is obtained from the `_receiver` address by calling the `.uniBTC()` function. However, the `_receiver` address refers to the `ccipPeer` contract address on the destination chain. This means that when retrieving the target address on the destination chain, the `_receiver` address on the destination chain is incorrectly used on the source chain.

```
    function estimateSendTokenFees(uint64 _destinationChainSelector, address _recipient, uint256
        _amount)
        external
        view
        returns (uint256)
    {
        address _receiver = allowlistedDestinationChains[_destinationChainSelector];
        require(_receiver != address(0), "USR007");
        address _target = CCIPPeer(payable(_receiver)).uniBTC();
```

Listing 2.6: ccip/src/ccipPeer.sol

Impact The target address for bridging `uniBTC` tokens is incorrect.

Suggestion Refactor the code logic.

2.2 Defi Security

2.2.1 Potential read-only reentrancy risk

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `DelayRedeemRouter` contract updates its states following a native token transfer, introducing a potential risk of read-only reentrancy.

```
function _claimDelayedRedeems(
    address recipient,
    uint256 maxNumberOfDelayedRedeemsToClaim
) internal {
    uint256 delayedRedeemsCompletedBefore = _userRedeems[recipient]
        .delayedRedeemsCompleted;
    uint256 numToClaim = 0;
    DebtTokenAmount[] memory debtAmounts;
    (numToClaim, debtAmounts) = _getDebtTokenAmount(
        recipient,
        delayedRedeemsCompletedBefore,
        redeemDelayTimestamp,
        maxNumberOfDelayedRedeemsToClaim
    );

    if (numToClaim > 0) {
        // mark the i delayedRedeems as claimed
        _userRedeems[recipient].delayedRedeemsCompleted =
            delayedRedeemsCompletedBefore +
            numToClaim;

        // transfer the delayedRedeems to the recipient
        uint256 burn_amount = 0;
        bytes memory data;
        for (uint256 i = 0; i < debtAmounts.length; i++) {
            address token = debtAmounts[i].token;
            uint256 amountUniBTC = debtAmounts[i].amount;
            uint256 amountToSend = _amounts(token, amountUniBTC);
            if (token == NATIVE_BTC) {
                // transfer native token to the recipient
                IVault(vault).execute(address(this), "", amountToSend);
                payable(recipient).transfer(amountToSend);
            } else {
                data = abi.encodeWithSelector(
                    IERC20.transfer.selector,
                    recipient,
                    amountToSend
                );
                // transfer erc20 token to the recipient
                IVault(vault).execute(token, data, 0);
            }
            tokenDebts[token].claimedAmount += amountUniBTC;
            burn_amount += amountUniBTC;
            emit DelayedRedeemsClaimed(recipient, token, amountToSend);
        }
        //burn claimed amount unibtc
        if (IERC20(uniBTC).allowance(address(this), vault) < burn_amount) {
            IERC20(uniBTC).safeApprove(vault, burn_amount);
        }
        data = abi.encodeWithSelector(
```

```
                IMintableContract.burnFrom.selector,
                address(this),
                burn_amount
            );
            IVault(vault).execute(uniBTC, data, 0);

            emit DelayedRedeemsCompleted(
                recipient,
                burn_amount,
                delayedRedeemsCompletedBefore + numToClaim
            );
        }
    }
```

Listing 2.7: contracts/contracts/proxies/stateful/redeem/DelayRedeemRouter.sol

Impact May introduce unexpected consequences for contract integrators.

Suggestion Revise the code accordingly.

2.3 Additional Recommendation

2.3.1 Remove unnecessary contract inheritance

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `Vault` contract inherits `PausableUpgradeable` but does not implement the associated `pause`/`unpause` functions. Instead, it uses a custom pausing mechanism within the `mint` functions at the token level, making this inheritance redundant and removable.

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

Listing 2.8: contracts/contracts/Vault.sol

Similarly, the `BR` contract inherits both the `ERC20` and `ERC20Burnable` contracts. However, since the `ERC20Burnable` contract already inherits `ERC20`, the direct inheritance from `ERC20` is redundant.

```
contract Bedrock is ERC20, ERC20Burnable, AccessControl {
```

Listing 2.9: contracts/contracts/BR.sol

Impact N/A

Suggestion Remove unnecessary contract inheritances.

2.3.2 Add sanity checks when setting crucial variables

Status Fixed in `Version 2`

Introduced by Version 1

Description There are no sanity checks when setting critical variables. To prevent misconfiguration, it is recommended to implement appropriate validation checks.

```
function initialize(address _defaultAdmin, address _uniBTC, address _sysSigner) external
    initializer {
    __AccessControl_init();
    _grantRole(DEFAULT_ADMIN_ROLE, _defaultAdmin);
    _grantRole(PAUSER_ROLE, _defaultAdmin);
    uniBTC = _uniBTC;
    sysSigner = _sysSigner;
    // uniBTC has 8 digital decimal, 20*1e5 = 0.02000000
    minTransferAmt = 20 * 1e5;
}
```

Listing 2.10: ccip/src/ccipPeer.sol

```
function setSysSinger(address _sysSigner) external onlyRole(DEFAULT_ADMIN_ROLE) {
    sysSigner = _sysSigner;
    emit SysSignerChange(_sysSigner);
}
```

Listing 2.11: ccip/src/ccipPeer.sol

```
function initialize(address _defaultAdmin, address _directBTC, address _vault, address _uniBTC)
     initializer public {
    require(_directBTC != address(0x0), "SYS001");
    require(_vault != address(0x0), "SYS001");
    require(_uniBTC != address(0x0), "SYS001");

    __AccessControl_init();
    _grantRole(DEFAULT_ADMIN_ROLE, _defaultAdmin);
    _grantRole(APPROVER_ROLE, _defaultAdmin);
    _grantRole(L1_MINTER_ROLE, _defaultAdmin);

    directBTC = _directBTC;
    vault = _vault;
    uniBTC = _uniBTC;
}
```

Listing 2.12: contracts/contracts/proxies/stateful/directBTC/DirectBTCMinter.sol

```
function _setRedeemDelayTimestamp(uint256 newValue) internal {
    require(newValue <= MAX_REDEEM_DELAY_DURATION_TIME, "USR012");
    emit redeemDelayTimestampSet(redeemDelayTimestamp, newValue);
    redeemDelayTimestamp = newValue;
}
```

Listing 2.13: contracts/contracts/proxies/stateful/redeem/DelayRedeemRouter.sol

Impact May lead to unexpected results.

Suggestion Add sanity checks when setting key parameters.

2.3.3 Ajust the order of checks to optimize gas usage

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When sending tokens, the `ccipPeer` first verifies the validity of the provided signature before checking whether the signature has been used. Since verifying the signature's validity consumes significant gas, rearranging the order of checks is more gas-efficient in most cases.

```
function sendToken(uint64 _destinationChainSelector, address _recipient, uint256 _amount, bytes
    memory _signature)
    external
    payable
    whenNotPaused
    validateReceiver(_recipient)
    returns (bytes32 messageId)
{
    require(_amount >= minTransferAmt, "USR006");
    require(
        verifySendTokenSign(msg.sender, _destinationChainSelector, _recipient, _amount,
            _signature), "SIGNERROR"
    );
    if (processedSignature[_signature]) revert SignatureProcessed();
    processedSignature[_signature] = true;
    return _sendToken(_destinationChainSelector, _recipient, _amount);
}
```

Listing 2.14: ccip/src/ccipPeer.sol

Impact N/A

Suggestion Refactor the order of checks accordingly.

2.3.4 Read `mBTC` address from the `mTokenSwap`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `MBTCProxy` is used to swap `mBTC` for `BTC`. The `mBTC` is defined as a contract variable; however, it is recommended to read it from the `mTokenSwap` contract to ensure consistency.

```
address public constant mBTC = 0xB880fd278198bd590252621d4CD071b1842E9Bcd;
address public constant mTokenSwap = 0x72A817715f174a32303e8C33cDCd25E0dACfE60b;
```

Listing 2.15: contracts/contracts/proxies/MBTCProxy.sol

Impact N/A

Suggestion Refactor the code accordingly.

2.3.5 Emit events when changing crucial parameters

Status Fixed in Version 2

Introduced by Version 1

Description When setting crucial parameters, the contracts should emit corresponding events to enhance maintainability.

```
function addToWrapBtcList(
    address _token
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    wrapBtcList[_token] = true;
}
```

Listing 2.16: contracts/contracts/proxies/stateful/redeem/DelayRedeemRouter.sol

```
function _setWhitelistEnabled(bool newValue) internal {
    whitelistEnabled = newValue;
}
```

Listing 2.17: contracts/contracts/proxies/stateful/redeem/DelayRedeemRouter.sol

Impact N/A

Suggestion Add events for changes to critical parameters.

2.3.6 Remove redundant checks for parameters

Status Fixed in Version 2

Introduced by Version 1

Description Checks on configuration variables should not be included in contexts that do not involve setting them. Since the values of configuration variables are determined after they are set, any comparison made prior to this is meaningless.

```
function _claimPrincipals(
    address recipient,
    uint256 maxNumberOfDelayedRedeemsToClaim
) internal {
    require(redeemPrincipalDelayTimestamp > redeemDelayTimestamp, "USR019");
    uint256 delayedRedeemsCompletedBefore = _userRedeems[recipient]
        .delayedRedeemsCompleted;
    uint256 numToClaim = 0;
    DebtTokenAmount[] memory debtAmounts;
```

Listing 2.18: contracts/contracts/proxies/stateful/redeem/DelayRedeemRouter.sol

Impact N/A

Suggestion Remove the redundant checks.

2.4 Note

2.4.1 Potential centralization risks

Introduced by `Version 1`

Description In the UniBTC contracts, several functions possess the privilege to set key parameters. Modifying these parameters can significantly affect the contracts' functionality, potentially rendering them unusable or causing them to enter an incorrect state

2.4.2 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 for the exact same amount. This allows users to *swap* supported tokens at a **1:1** ratio through the `Vault` contract. If one of the supported tokens becomes unpegged from the underlying `BTC`, it can still be exchanged for other tokens at a 1:1 ratio. This may deplete the `Vault` contract and lead to other unexpected consequences.

2.4.3 The slippage check is applicable only to BTC-pegged tokens

Introduced by `Version 1`

Description The `SwapProxy` allows the owner to swap tokens with slippage checks; however, these checks apply only to BTC-pegged tokens. Specifically, the slippage check requires the output amount of the swap to exceed a predefined ratio of the input amount, which is valid only when the prices of the two tokens are close to a 1:1 ratio.

```
function _swap(uint256 amountIn, uint256 slippage, address pool) internal {
    require(_poolsInfo[pool].isValid, "USR021");
    uint256 amountOutMin = (amountIn * (SLIPPAGE_RANGE - slippage)) /
        SLIPPAGE_RANGE;
    uint256 vaultToTokenBalanceBefore = IERC20(toToken).balanceOf(
        bedrockVault
    );
    if (_poolsInfo[pool].protocol == UNISWAP_V3_PROTOCOL) {
        _swapByUniswapV3Router2(amountIn, amountOutMin, pool);
    } else if (_poolsInfo[pool].protocol == UNISWAP_V2_PROTOCOL) {
        _swapByUniswapV2Router2(amountIn, amountOutMin);
    } else if (_poolsInfo[pool].protocol == CURVE_PROTOCOL) {
        _swapByCurve(amountIn, amountOutMin, pool);
    }
    uint256 vaultToTokenBalanceAfter = IERC20(toToken).balanceOf(
        bedrockVault
    );
    uint256 amount = vaultToTokenBalanceAfter - vaultToTokenBalanceBefore;
    require(amount >= amountOutMin, "USR003");
    emit SwapSuccessAmount(toToken, amount);
}
```

Listing 2.19: contracts/contracts/proxies/SwapProxy.sol

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS