Ordinal Bridge Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

Bridge between BRC-20 & ERC-20. A gateway to Liquidity for BRC-20 tokens!

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Overview

Project	Ordinal Bridge
Repository	bridgecontract
Commit	87738d9112a4...
Date	June 2023

Issues Found

Severity	Count
Critical Risk	2
High Risk	1
Medium Risk	2
Low Risk	1
Informational	0
Gas Optimizations	0
Total Issues	**6**

Summary of Findings

Title	Status
[C-1] Users money will be lost forever if they dont use an taproot address	Resolved
[C-2] Oversupply vulnerability due to lack of maximum limit enforcement in ZUTToken contract	Resolved
[H-1] Decimal precision mismatch in ZUToken creation and burning may result in incorrect token exchange ratios	Resolved
[M-1] Risk of funds getting stuck due to unchecked result of fee recipient call	Resolved
[M-2] Lack of validation for duplicate transaction IDs in addMintERCEntries function	Resolved
[L-1] getBurnForBRCEntriesToProcess wil DoS due to OOG	Resolved

5 Findings

5.1 Critical Risk

[C-1] Users money will be lost forever if they dont use an taproot address

Context: TokenFactory.sol

Impact: The current implementation of the `burnERCTokenForBRC` function allows users to provide any BTC address for receiving BRC tokens. However, BRC tokens are only compatible with ordinal taproot addresses (P2TR). If a user provides a valid BTC address that is not an ordinal compatible taproot address, their funds will be permanently lost. This vulnerability poses a risk of financial loss for users.

Description: The `burnERCTokenForBRC` function is used to swap ERC's for BRC's. The user has to provide a ticker (ordinal identifier), amount, btcAddress and meta.

Once a swap is "initiated", there's no way for the user to cancel it or change the receiver address.

```
function burnERCTokenForBRC(
    string calldata ticker,
    uint256 amount,
    string calldata btcAddress,
    string calldata meta
) external payable nonReentrant whenNotPaused {

    string memory uppercaseTicker = uppercase(ticker);
    require(msg.value == BURN_ETH_FEE, "Incorrect fee");
    // solhint-disable-next-line
    feeRecipient.call{ value: BURN_ETH_FEE }("");
    require(tokenContracts[uppercaseTicker] != address(0), "Invalid ticker");

    ZUTToken(tokenContracts[uppercaseTicker]).burnFrom(msg.sender, amount);

    uint256 id = burnForBRCEntries.length;
    burnForBRCEntries.push(BurnForBRCEntry(id, uppercaseTicker, msg.sender, amount, btcAddress, meta, false));

    emit BurnForBRCEntryAdded(uppercaseTicker, msg.sender, btcAddress, id, amount, meta);
}
```

If we refer to BRC's docs, it clearly states:

> Do not send inscriptions to non ordinal compatible wallet taproot addresses

The vulnerability here stems from the fact that the user can set any kind of BTC address (P2PKH, P2SH, P2WPKH) while BRC's as of right now work only with P2TR.

As a consequence, if the user uses a valid. BTC address, but not an ordinal compatible taproot one, his funds will be forever lost.

Recommendation: To address this vulnerability, it is recommended to implement a validation step to ensure that the btcAddress provided by the user is an ordinal compatible taproot address (P2TR).

The following steps can be taken as a recommendation:

1. Before proceeding with the burn transaction, validate the btcAddress format.
2. Check that the btcAddress starts with "bc1p", which is the prefix for P2TR addresses.
3. If the btcAddress does not meet the required format, reject the transaction and provide an error message to the user, instructing them to use an ordinal compatible taproot address (P2TR).

```
function isOrdinalTaprootAddress(string memory btcAddress) internal pure returns (bool) {
    // Check that the btcAddress starts with "bc1p" (P2TR address prefix)
    bytes memory addressBytes = bytes(btcAddress);
    bytes memory prefixBytes = bytes("bc1p");

    for (uint i = 0; i < prefixBytes.length; i++) {
        if (addressBytes[i] != prefixBytes[i]) {
            return false;
        }
    }
    return true;
}

function burnERCTokenForBRC(
    string calldata ticker,
    uint256 amount,
    string calldata btcAddress,
    string calldata meta
) external payable nonReentrant whenNotPaused {
    // Validate that the btcAddress is an ordinal compatible taproot address (P2TR)
    require(isOrdinalTaprootAddress(btcAddress), "Invalid BTC address format");
    // Rest of the function implementation...
}
```

[C-2] Oversupply vulnerability due to lack of maximum limit enforcement in ZUTToken contract

Context: ZUTToken.sol

Impact: The current implementation of the ZUTToken contract allows minting an type(uint256).max amount of ZUTTokens, which can result in exceeding the maximum supply specified by the BRC documentation. This vulnerability poses a risk of generating ZUTTokens that cannot be converted to ordinals.

Description: When users "bridge" a ordinal, they are going to receive a ZUTToken in exchange.

```
contract ZUTToken is Ownable, ERC20 {
    using Strings for string;

    constructor(string memory name) ERC20(concat("Wrapped-", name), concat("w", name)) {}

    function concat(string memory a, string memory b) internal pure returns (string memory) {
        return string(abi.encodePacked(a, b));
    }

    function mintTo(address to, uint256 amount) external onlyOwner {
        _mint(to, amount);
    }

    function burnFrom(address to, uint256 amount) external onlyOwner {
        _burn(to, amount);
    }
}
```

The "allowance" is given by creating an ERC entry,

```
function addMintERCEntries(
        string[] calldata requestedBRCTickers,
        uint256[] calldata amounts,
        address[] calldata users,
        string[] calldata btcTxIds
    ) external onlyOwner {
```

And then the BRC -> ERC20 bridge happens here by minting ZUTTokens.

```
        ZUTToken(tokenContracts[uppercaseTicker]).mintTo(feeRecipient, feeTokenAmount);
        ZUTToken(tokenContracts[uppercaseTicker]).mintTo(msg.sender, userTokenAmount);
```

However, the ZUTToken contract does not implement a maximum supply check, thus allowing for an arbitrary amount of tokens to be minted by the contract owner. This is a critical issue because the equivalent BRC token has a documented maximum supply limit that cannot exceed `uint64_max`.

In the absence of a maximum supply enforcement in the `mintTo` function of the ZUTToken contract, an excess of ZUTTokens can be created. This oversupply is irreparable as the excess tokens can never be bridged back to the equivalent BRC token due to its supply limit.

Additionally, the `addMintERCEntries` function lacks checks to ensure the `amounts` array provided respects the maximum supply limit, allowing a potentially harmful oversupply when minting the ZUTTokens.

The bridge function that triggers the minting of ZUTTokens doesn't enforce any checks to prevent an oversupply, leading to an uncontrolled creation of tokens beyond the maximum supply limit of the equivalent BRC token.

Recommendation: Enforce a maximum supply limit in the ZUTToken contract to match the `uint64_max` limit of the equivalent BRC token. This could be achieved by introducing a state variable to keep track of the total supply of ZUTTokens.

Modify the `mintTo` function to check if the minting of new tokens would exceed the maximum supply limit. If it would, the function should revert the transaction.

Apply similar checks in the `addMintERCEntries` function and the bridge function to prevent minting of tokens beyond the maximum limit.

For example:

```
    uint256 public maxSupply;

    constructor(string memory name) ERC20(concat("Wrapped-", name), concat("w", name)) {
        // Set the max supply to the maximum uint64 value.
        maxSupply = 2**64 - 1;
    }

  function mintTo(address to, uint256 amount) external onlyOwner {
        require(totalSupply().add(amount) <= maxSupply, "Cannot exceed maximum supply");
        _mint(to, amount);
    }
```

5.2 High Risk

[H-1] Decimal precision mismatch in ZUToken creation and burning may result in incorrect token exchange ratios

Context: TokenFactory.sol

Impact: The vulnerability in the code allows for a decimal precision mismatch between the ZToken (ERC) created and the BRC token being swapped. This mismatch can lead to incorrect token exchange ratios and potential financial gain or loss for users.

Description: The claimERCEntryForWallet function is used by users who want to claim the ERC they have been given in exchange for the BRC they have swapped.

```
function claimERCEntryForWallet(string memory ticker) external whenNotPaused nonReentrant {

    string memory uppercaseTicker = uppercase(ticker);
    require(mintableERCTokens[uppercaseTicker][msg.sender] > 0, "No entry");
    if (tokenContracts[uppercaseTicker] == address(0)) {

        ZUTToken token = new ZUTToken(uppercaseTicker);
        tokenContracts[uppercaseTicker] = address(token);
        emit ERCTokenContractCreated(uppercaseTicker, address(token));
    }
    //TODO check value
    uint256 feeTokenAmount = (mintableERCTokens[uppercaseTicker][msg.sender] * TOKEN_FEE_PERCENT) /
    ↪   100;
    // @audit fee could be 0

    uint256 userTokenAmount = mintableERCTokens[uppercaseTicker][msg.sender] - feeTokenAmount;

    ZUTToken(tokenContracts[uppercaseTicker]).mintTo(feeRecipient, feeTokenAmount);
    ZUTToken(tokenContracts[uppercaseTicker]).mintTo(msg.sender, userTokenAmount);

    emit MintableERCEntryClaimed(uppercaseTicker, msg.sender, mintableERCTokens[uppercaseTicker][msg.sender],
    ↪   userTokenAmount, feeTokenAmount);

    delete mintableERCTokens[uppercaseTicker][msg.sender];
}
```

The following check `if (tokenContracts[uppercaseTicker] == address(0)) {` is used to check if there's an already associated "ZUTToken" with the afforemented BRC ticker. The ticker is the 4 letter identifier of the brc-20.

Then, if there's no "ERC" token for that BRC, it proceeds by creating a new token. The vulnerability in this approach is that the ZToken is created by default with 18 decimals, whereas the BRC token could have different amount of decimals.

As a consequence, when the user burns his ZTokens, he would be eligible for more/less tokens if the BRC tokens are different.

Consider the following scenario:

There's an BRC20 with the following properties:

```
"p": ...,
"op": ...,
"tick": "b032",
"max": ...
"dec": 8
```

Then Amelie wants to swap some `b032`, hence the admins call`addMintERCEntries` for b032 tick with 1e8 as amount. However, when the user claims it through `claimERCEntryForWallet` he will receive ZUTToken in 1e18.

The problem is, that when Amelie decides to swap back for `b032`, she would be getting the amount in 18 decimals instead of 8 decimals.

```
function burnERCTokenForBRC(
    string calldata ticker,
    uint256 amount,
    string calldata btcAddress,
    string calldata meta
) external payable nonReentrant whenNotPaused {

    string memory uppercaseTicker = uppercase(ticker);
    require(msg.value == BURN_ETH_FEE, "Incorrect fee");
    // solhint-disable-next-line
    feeRecipient.call{ value: BURN_ETH_FEE }("");
    require(tokenContracts[uppercaseTicker] != address(0), "Invalid ticker");

    ZUTToken(tokenContracts[uppercaseTicker]).burnFrom(msg.sender, amount);

    uint256 id = burnForBRCEntries.length;
    burnForBRCEntries.push(BurnForBRCEntry(id, uppercaseTicker, msg.sender, amount, btcAddress, meta, false));

    emit BurnForBRCEntryAdded(uppercaseTicker, msg.sender, btcAddress, id, amount, meta);
}
```

As a result, her exchange ratio will be heavily inflated

Recommendation: To address the vulnerability and ensure accurate token exchange ratios, the following recommendations are provided:

1. Modify the `claimERCEntryForWallet` function to retrieve the decimal precision of the BRC token associated with the ticker. This can be achieved by adding a mapping or storage to keep track of the decimal precision for each BRC token.

```
// Add a mapping to store decimal precision for each BRC token
mapping(string => uint8) public brcTokenDecimalPrecision;
```

2. Use the obtained decimal precision to initialize the ZToken (ERC) with the correct decimal precision when creating a new token in the `claimERCEntryForWallet` function. This ensures that the ZToken aligns with the intended precision of the BRC token.

```
// Set the decimal precision of the ZToken based on the BRC token
uint8 decimalPrecision = brcTokenDecimalPrecision[uppercaseTicker];
ZUTToken token = new ZUTToken(uppercaseTicker, decimalPrecision);
```

5.3 Medium Risk

[M-1] Risk of funds getting stuck due to unchecked result of fee recipient call

Context: TokenFactory.sol

Impact: This vulnerability exposes the smart contract to a potential loss of funds. If the `feeRecipient.call` statement in the `burnERCTokenForBRC` function returns `false`, the function will proceed with the execution without paying the fee recipient. As a result, the `BURN_ETH_FEE` will remain in the contract indefinitely and become irretrievable.

Description: When burning ERC for BRC, there's a fee that is due to the fee recipient. The vulnerability lies in the lack of validation for the result of the `feeRecipient.call` function call. The code does not check whether the call was successful or if it returns `false`. Consequently, even if the call fails, the function execution continues, allowing the contract to proceed without paying the fee recipient.

In 99% of the cases, this would be a minor issue, but what makes it a medium is that if this happens `BURN_ETH_FEE` won't be retrievable and will be forever stuck in the contract.

```
function burnERCTokenForBRC(
        string calldata ticker,
        uint256 amount,
        string calldata btcAddress,
        string calldata meta
    ) external payable nonReentrant whenNotPaused {

        string memory uppercaseTicker = uppercase(ticker);
        require(msg.value == BURN_ETH_FEE, "Incorrect fee");
        // solhint-disable-next-line
        feeRecipient.call{ value: BURN_ETH_FEE }("");
        require(tokenContracts[uppercaseTicker] != address(0), "Invalid ticker");

        ZUTToken(tokenContracts[uppercaseTicker]).burnFrom(msg.sender, amount);

        uint256 id = burnForBRCEntries.length;
        burnForBRCEntries.push(BurnForBRCEntry(id, uppercaseTicker, msg.sender, amount, btcAddress, meta, false));

        emit BurnForBRCEntryAdded(uppercaseTicker, msg.sender, btcAddress, id, amount, meta);
    }
```

Recommendation: To address this vulnerability, it is recommended to implement proper validation of the result returned by the `feeRecipient.call` statement. By checking the return value, the function can handle cases where the call fails or returns `false`. Below is an updated version of the code with the recommended changes:

```
function burnERCTokenForBRC(
    string calldata ticker,
    uint256 amount,
    string calldata btcAddress,
    string calldata meta
) external payable nonReentrant whenNotPaused {
    string memory uppercaseTicker = uppercase(ticker);
    require(msg.value == BURN_ETH_FEE, "Incorrect fee");

    // Ensure the fee recipient call is successful
    (bool success, ) = feeRecipient.call{ value: BURN_ETH_FEE }("");
    require(success, "Fee recipient call failed");

    require(tokenContracts[uppercaseTicker] != address(0), "Invalid ticker");

    ZUTToken(tokenContracts[uppercaseTicker]).burnFrom(msg.sender, amount);

    uint256 id = burnForBRCEntries.length;
    burnForBRCEntries.push(BurnForBRCEntry(id, uppercaseTicker, msg.sender, amount, btcAddress, meta, false));

    emit BurnForBRCEntryAdded(uppercaseTicker, msg.sender, btcAddress, id, amount, meta);
}
```

[M-2] Lack of validation for duplicate transaction IDs in addMintERCEntries function

Context: TokenFactory.sol

Impact: The lack of validation for duplicate transaction IDs in the `addMintERCEntries` function can lead to the repeated addition of ERC20 entries with the same `btcTxId` value. This vulnerability can potentially result in the creation of multiple duplicate entries, leading to inaccurate or misleading data within the system. However, since this function is admin-controlled, the impact is rated as medium.

Description: The `addMintERCEntries` function allows owners to add mintable ERC20 entries by providing various parameters, including the `btcTxIds` array. However, the current implementation lacks validation to check if a transaction ID has already been used. This means that the same `btcTxId` can be used multiple times to add ERC20 entries without any restrictions.

```
function addMintERCEntries(
        string[] calldata requestedBRCTickers,
        uint256[] calldata amounts,
        address[] calldata users,
        string[] calldata btcTxIds
    ) external onlyOwner {
        require(
            requestedBRCTickers.length > 0 &&
                requestedBRCTickers.length == amounts.length &&
                requestedBRCTickers.length == users.length &&
                requestedBRCTickers.length == btcTxIds.length,
            "Invalid params"
        );
        for (uint256 index = 0; index < requestedBRCTickers.length; index++) {

            string memory uppercaseTicker = uppercase(requestedBRCTickers[index]);
            mintableERCTokens[uppercaseTicker][users[index]] += amounts[index];
            if (brctokenTickerMap[uppercaseTicker] == false) {
                brcTickers.push(uppercaseTicker);
                brctokenTickerMap[uppercaseTicker] = true;
            }

            emit MintableERCEntryAdded(uppercaseTicker, users[index], amounts[index], btcTxIds[index]);
        }
    }
```

As a result, duplicate entries with the same `btcTxId` can be added to the system, potentially causing data integrity issues.

Without proper validation, it becomes difficult to differentiate between unique transactions and repeated submissions. This vulnerability can lead to inaccurate reporting, incorrect balance calculations, and confusion regarding the actual state of the system.

Recommendation: To address this vulnerability, it is recommended to implement validation checks to ensure that each `btcTxId` provided in the `addMintERCEntries` function is unique and has not been previously used. This can be achieved by maintaining a mapping or a data structure to store and verify the uniqueness of transaction IDs.

Consider implementing the following steps to mitigate the vulnerability:

1. Create a mapping or data structure to store used `btcTxIds` and their corresponding status (e.g., `bool` value indicating whether the ID has been used or not).
2. Before processing each `btcTxId` within the loop, check if it has already been used by querying the mapping or data structure created in the previous step.
3. If the `btcTxId` is found to be already used, skip the entry or raise an error to indicate the duplication. Ensure proper error handling and informative error messages are provided to assist in debugging
4. If the `btcTxId` is unique and has not been used, proceed with adding the ERC20 entry as before.

```
// Define a mapping to store used btcTxIds and their status
mapping(string => bool) private usedBtcTxIds;

function addMintERCEntries(
    string[] calldata requestedBRCTickers,
    uint256[] calldata amounts,
    address[] calldata users,
    string[] calldata btcTxIds
) external onlyOwner {
    require(
        requestedBRCTickers.length > 0 &&
        requestedBRCTickers.length == amounts.length &&
        requestedBRCTickers.length == users.length &&
        requestedBRCTickers.length == btcTxIds.length,
        "Invalid params"
    );
    for (uint256 index = 0; index < requestedBRCTickers.length; index++) {
        string memory uppercaseTicker = uppercase(requestedBRCTickers[index]);

+         // Check if btcTxId has already been used
+         require(!usedBtcTxIds[btcTxIds[index]], "Duplicate btcTxId");

+         // Update the mapping to mark the btcTxId as used
+         usedBtcTxIds[btcTxIds[index]] = true;

        mintableERCTokens[uppercaseTicker][users[index]] += amounts[index];
        if (brctokenTickerMap[uppercaseTicker] == false) {
            brcTickers.push(uppercaseTicker);
            brctokenTickerMap[uppercaseTicker] = true;
        }

        emit MintableERCEntryAdded(uppercaseTicker, users[index], amounts[index], btcTxIds[index]);
    }
}
```

5.4 Low Risk

[L-1] getBurnForBRCEntriesToProcess wil DoS due to OOG

Context: TokenFactory.sol

Impact: The vulnerability in the `getBurnForBRCEntriesCountToProcess` and `getTickersAndTokenAddresses` functions can lead to denial of service (DoS) attacks due to out of gas, rendering these functions and the dependent function `getBurnForBRCEntriesToProcess` unusable.

Description: The issue stems from the unbounded growth of the `burnForBRCEntries` and `brcTickers` arrays. The vulnerable code snippets responsible for the vulnerability are as follows:

```
    if (brctokenTickerMap[uppercaseTicker] == false) {
        brcTickers.push(uppercaseTicker);
```

```
uint256 id = burnForBRCEntries.length;
burnForBRCEntries.push(BurnForBRCEntry(id, uppercaseTicker, msg.sender, amount, btcAddress, meta, false));
```

In the first code snippet, the `brcTickers` array grows without any limits, as it keeps appending values to the end of the array. Similarly, the second code snippet leads to unbounded growth of the `burnForBRCEntries` array.

As a consequence, the `getBurnForBRCEntriesCountToProcess` function, which relies on the length of the `burnForBRCEntries` array, will become increasingly inefficient and consume excessive gas as the array grows. This vulnerability not only impacts gas costs but also makes the `getBurnForBRCEntriesToProcess` function, which depends on `getBurnForBRCEntriesCountToProcess`, inoperable due to the increasingly unmanageable arrays.

The same vulnerability applies to `checkPendingERCToClaimForWallet` as well

Recommendation: Consider using a circular buffer/mapping or similar data structure to efficiently manage or remove of old entries. This allows for constant-time removal of the oldest entry while maintaining a fixed-size array.