Audit Report

BRZ Bridge – Ethereum Smart Contracts

November 6, 2021

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of this Report

Oak Security has been engaged by TransferoSwiss to perform a security audit of the BRZ bridge smart contracts.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behaviour.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following GitHub repository:

https://github.com/TransferoSwiss/brz-token-bridge

Commit hash: `e84a1eb6ad140ce0bedd292ff64eec3bc25e7fe0`

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The smart contracts implement a trusted bridge that allows moving BRZ tokens between Ethereum-like blockchains.

Important note: The BRZ is a trusted bridge relying on a monitor process to relay transactions. This monitor process is a centralized entity that relies on a private key used as a hot wallet. In the case of the monitor service being compromised, the attacker could gain access to all the funds managed by the bridge. **The actual monitor process was not part of the scope of this audit.**

How to read this Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged** or **Resolved**. Informational notes do not have a status, since we consider them optional recommendations.

Note, that audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note, that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Summary of Findings

No	Description	Severity	Status
1	Bridge operator has full control over funds and relies on backend service controlling a hot wallet	Major	Acknowledged
2	Addresses are encoded as string and not validated	Minor	Resolved
3	Adding of new blockchains can be front-run to avoid paying the minimal fee	Minor	Resolved
4	Unused parameter	Minor	Resolved
5	Outdated dependencies in build and deployment system	Minor	Resolved
6	Inefficient array data structure for tracking supported blockchain	Informational	-
7	Potential data structure optimization	Informational	-
8	Oracle functionality mixed with bridge logic	Informational	-
9	Admin role also controls oracle	Informational	-
10	Slightly outdated OpenZeppelin release used	Informational	-
11	Unnecessary use of modifier for authorization on private method	Informational	-
12	Unnecessary long digit constant	Informational	-
13	Gas Optimizations	Informational	-

Code Quality Criteria

Criteria	Status	Comment
Code complexity	Low	-
Code readability and clarity	High	-
Level of Documentation	High	-
Test Coverage	Medium-High	-

Detailed Findings

1. Bridge operator has full control over funds and relies on backend service controlling a hot wallet

Severity: Major

The centralized design of the bridge allows the bridge operator full access to the users' funds. This means the operating entity has to be fully trusted since it can withdraw users' funds and censor transactions.

The design comes with additional security risks in the form of a backend process (MONITOR) that manages the bridge through a single private key, which is used as a hot wallet.
In case of the server being compromised and an attacker gaining access to the key, all funds are at risk, and transactions can be censored or executed at will. Furthermore, a potential DoS attack on the MONITOR could block the entire bridge functionality.

Recommendation

One way of reducing risk is to use multiple MONITORs or relayers with an additional voting/threshold mechanism to improve the overall bridge security.
(Example for a bridge with multiple relayers https://github.com/ChainSafe/ChainBridge).

An alternative would be to use a multi-sig like Gnosis Safe https://gnosis-safe.io/ as the monitor role with the current bridge implementation but require multiple signatures from different monitor nodes before executing an `acceptTransfer` transaction.

Both approaches are not very gas-efficient, a more efficient design would involve an off-chain multi-sig.

If the centralized design is kept, backend hardening measures should be taken to reduce the risk of the controlling key being compromised. A hardware security model for key management and/ or trusted execution environments can also increase security.
In addition, a key rotation protocol and key compromise protocol should be developed. Monitoring for suspicious activities can also help to reduce the risk.
One way to mitigate the impact of a key compromise is to have a separate role for emergency withdraws, managed by a cold multisig wallet. This works best in conjunction with a delay for large value bridge transactions above a certain threshold, to give the operator team time to proceed with emergency withdrawals.

Status: Acknowledged

Team reply: *“We acknowledge the issue since this risk is inherent to our business model.*

2. Addresses are encoded as string and not validated

Severity: Minor

The function `receiveTokens` takes a string parameter as the destination address. The reasoning behind this seems to be that some receiver blockchains use different address encoding. However, this means that addresses are not validated and tokens might be sent to an invalid address.

Recommendation

One option to deal with this is to track address encodings used in different blockchains and cast them to the `address` data type if appropriate. Alternatively, since the number of potential address formats are limited, address validation could be implemented on string or byte array data types. At the very least, basic checks, such as verifying the length of the destination address parameter could be performed.

Status: Resolved

3. Adding of new blockchains can be front-run to avoid paying the minimal fee

Severity: Minor

Adding and configuring the support for a new blockchain requires multiple transactions:
- `addBlockchain`
- `setMinorTokenAmount`
- `setMinGasPrice`

The function `receiveTokens` can be called immediately after the first `addBlockchain` transaction. Such a call would allow the usage of the bridge with a zero `minBRZFee`.

This attack might be worthwhile for an attacker if a user wants to use the bridge with a high fee.

Recommendation

Consider modifying the `addBlockchain` method to receive all required configuration parameters and initialize the newly supported blockchain in a single transaction. Alternatively, a proxy contract could be used to execute all individual function calls in one transaction.

Status: Resolved

4. Unused sender parameter in accept transfer function

Severity: Minor

In the `acceptTransfer` function, the `sender` parameter is passed but not included in the transaction id calculation. Whilst the sender information does not add security, it might add value to the calculation of the transaction id for off-chain purposes.

Recommendation

Consider including sender information in the transaction id calculation or remove the unnecessary parameter.

Status: Resolved

5. Outdated dependencies in build- and deployment system

Severity: Minor

The build- and deployment system has several outdated dependencies with known security vulnerabilities. Some of these relate to cryptographic primitives used for deployment.

Recommendation

Run `npm audit` and update dependencies.

Status: Resolved

6. Inefficient array data structure for tracking supported blockchains

Severity: Informational

The different blockchains supported are stored in an array of strings. This requires an iteration of the entire list in functions like `existsBlockchain`.

Recommendation

The reason for the list implementation seems to be the `listBlockchain` function. If the entire list is not required on-chain it could be calculated based on emitted events off-chain.
If the list is required on-chain, using a `uint` data type for ids (starting from 0) together with mappings and a counter could improve the performance.
The replacement of the type `string` with `bytes32` for the blockchain id would already decrease the gas usage.

7. Potential data structure optimization

Severity: Informational

```
mapping(string => uint256) private minBRZFee;
mapping(string => uint256) private minGasPrice;
mapping(string => uint256) private minTokenAmount;
```

Recommendation

The blockchain-related information could be stored in one mapping using a Struct. This would allow storage slot optimization to reduce gas usage. If smaller variables than `uint256` are used they can be bundled into groups of `uint256` variables.

8. Oracle functionality mixed with bridge logic

Severity: Informational

The `minBRZFee` variable is required to calculate fees in the `receiveTokens` function. The calculation happens on-chain based on `quoteETH_BRZ`, `gasAcceptTransfer` and `minGasPrice` in the `_updateMinBRZFee` function. Currently, these variables are set by admin calls. In future versions, the `quoteETH_BRZ` will be provided by oracles, according to the code comments. The contract is already laid out to include oracle support in a future version. However, integrating oracle functionality into the bridge module itself is not considered best practice due to poor separation of concerns.

Recommendation

We recommend optimizing the smart contract for the current use case. The variable `minBRZFee` itself could be set by an external call. There seems to be no need to perform the calculation of `minBRZFee` on-chain.

Once oracles become necessary in future versions, we recommend using a separate contract for this, since it improves modularity and allows substituting the oracle more easily.

Both recommendations would reduce the current code complexity.

9. Admin role also controls oracle

Severity: Informational

The information required to calculate `minBRZFee` is currently provided by the admin role. This concentrates a lot of functionality in a single private key managed by the admin server (MONITOR).

Recommendation

A new oracle role would provide a better separation of concerns. Only the oracle role would be allowed to change fee-related information. This would also reduce the impact of key compromise by splitting functionality between keys.

10. Slightly outdated OpenZeppelin release used

Severity: Informational

The codebase imports a relatively recent version of the OpenZeppelin smart contract library. However, there have been recent security releases that fix vulnerabilities. These issues seem not to apply in the present use case of the contracts. However, we recommend using the latest security release.

Recommendation

Consider updating OpenZeppelin.

11. Unnecessary use of modifier for authorization on private method

Severity: Informational

The private `_processTransaction` method is covered by access control modifiers.

Authorization modifiers like `onlyMonitor` or `whenNotPaused` are only required for external or public methods but not for private ones. In this particular case, they are already covered in the caller function.

Recommendation

Consider removing the unnecessary modifiers to optimize the call and increase code clarity.

12. Unnecessary long digit constant

Severity: Informational

A constant is defined to represent 10^18 for token decimal conversion:

```
uint256 public constant ETH_IN_WEI = 1000000000000000000;
```

Recommendation

Consider using the built-in `ether` to represent 10^18 keyword:

```
uint256 public constant ETH_IN_WEI = 1 ether;
```

13. Gas Optimizations

Severity: Informational

The contracts can be optimized for more efficient gas usage in multiple places.

Recommendations

Removal of getter methods

A common pattern in the code is to keep variables private together with an additional getter method. For example like `totalFeeReceivedBridge` and `getTotalFeeReceivedBridge`.

It would be more gas efficient to make the variable public and remove the getter method. A public variable has by default a getter method already.

More efficient `delBlockchain` implementation

The gas inefficient `existsBlockchain` call can be avoided, as in the below code example, and an additional index variable is unnecessary.

```
function delBlockchain(string memory name)
  external
  onlyOwner
  whenNotPaused
  returns (bool)
{

  require(blockchain.length > 1, "Bridge: requires at least 1 blockchain");
  uint256 index = 0;
  for (; index < blockchain.length; index++) {
    if (compareStrings(name, blockchain[index])) {
      break;
    }
  }
  require(index < blockchain.length, "Bridge: blockchain does not exists");

  blockchain[index] = blockchain[blockchain.length - 1];
  blockchain.pop();
  return true;
}
```

However, if other recommendations are implemented a loop iteration might not be required anymore.

Emit Events as the last step in a function

Example: `receiveTokens` function

The method could still fail in the `transferFrom` call. There would be less gas spent in that case if the emit event happens as the last step in the method.

Requires as the first step

It is more efficient to perform checks using `require` statements as early as possible, i.e. as soon as the values are available. For input validation, this is at the beginning of the function.