# Security Audit Report for ExoSwap Bridge Contracts

**Date:** July 3, 2023

**Version:** 1.0

**Contact:** contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.07.10 15:06:56 +08'00'

# Contents

## Report Manifest

| Item | Description |
|---|---|
| Client | ExoSwap |
| Target | ExoSwap Bridge Contracts |

## Version History

| Version | Date | Description |
|---|---|---|
| 1.0 | July 3, 2023 | First Release |

**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 5 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 target of this audit is the code repo of ExoSwap Bridge Contracts [1] of the ExoSwap project. The ExoSwap project is a cross-chain bridge that bridges assets over multiple EVM-based chains.

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 |
|---|---|---|
| ExoSwap Contracts | `Version 1` | `22400563e5cb685ddd68fea3485673e38fcfd50d` |
| | `Version 2` | `f51806bc48a1da04de6d709196c6e2ef7d8f01d0` |

## 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.

## 1.3 Procedure of Auditing

We perform the audit according to the following procedure.

[1] https://github.com/exoswapio/monorepo

- **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 find **four** potential issues. We have **one** recommendation.

- High Risk: 3
- Low Risk: 1
- Recommendation: 1

| ID | Severity | Description | Category | Status |
|---|---|---|---|---|
| 1 | High | Invalid multi-signature verification | DeFi Security | Fixed |
| 2 | Low | Potential access control takeover in FeeManager | DeFi Security | Fixed |
| 3 | High | Potential price manipulation | DeFi Security | Fixed |
| 4 | High | Centralization Risk | DeFi Security | Fixed |
| 5 | - | Require minimum number of cosigners | Recommendation | Acknowledged |

The details are provided in the following sections.

## 2.1 DeFi Security

### 2.1.1 Invalid multi-signature verification

**Severity** High

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** In the "exit" process, multiple signatures of the exit data by two-thirds of all the cosigners are required. This logic is implemented in the verify function of the BridgeCosignerManager contract.

However, this logic is implemented incorrectly. As shown in the following code segment, there is a cached variable that stores the cumulative cosigners. Inside the for loop, the _inCache is invoked to check if the signer has appeared. However, the _inCache is implemented incorrectly. For example, if two signatures are required (i.e. the _required variable equals to $2$), and there are three signatures in the signatures parameter with the first signature signed by a non-signer (but this signature is still valid). Then the cached[0] == address(0) is always true, and the remaining two signatures can duplicate, so only one valid signature is needed.

```
function verify(
    bytes32 commitment,
    uint256 chainId,
    bytes[] calldata signatures
) external view override returns (bool) {
    uint8 _required = getCosignCount(chainId);
    if (_required > signatures.length) {
        return false;
    }

    address[] memory cached = new address[](signatures.length);
    uint8 signersMatch;
```

```
    for (uint8 i = 0; i < signatures.length; i++) {
        address signer = recover(commitment, signatures[i]);
        Cosigner memory cosigner = _cosigners[signer];

        if (
            cosigner.active &&
            cosigner.chainId == chainId &&
            !_inCache(cached, signer)
        ) {
            signersMatch++;
            cached[i] = signer;
            if (signersMatch == _required) return true;
        }
    }

    return false;
}

function _inCache(
    address[] memory cached,
    address signer
) internal pure returns (bool hasCache) {
    for (uint8 j = 0; j < cached.length; j++) {
        if (cached[j] == signer) {
            hasCache = true;
            break;
        }
        // prevent iteration if cache not updated in slot
        if (cached[j] == address(0)) {
            break;
        }
    }
}
```

**Listing 2.1:** BridgeCosignerManager.sol

**Impact** The multi-signature verification logic is broken. Only one valid signature is used to pass the verification.

**Suggestion** Fix the multi-signature verification logic.

### 2.1.2 Potential access control takeover in `FeeManager`

**Severity** Low

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** The access control in the `FeeManager` contract is in a “first come first served” manner. As shown in the following code segment, the `reserveFee` function assign the first caller of a specific `AppId` to the owner of the `AppId`. Therefore, the registration of the owner of a specific `AppId` can be front-run and the fee configuration for this `AppId` can be modified.

```
function reserveFee(
    bytes32 _appId,
    uint256 _chainId,
    uint256 _baseFee,
    uint256 _feePerByte
) public override {
    require(_appId != DEFAULT_APP_ID, Errors.R_RESERVED_ENTITY);
    bytes32 key = getAppOwnerKey(_appId);
    require(!_appOwners[key], Errors.R_RESERVED_OWNER);
    _appOwners[key] = true;
    _reserveFee(_appId, _chainId, _baseFee, _feePerByte);
}
```

**Listing 2.2:** FeeManager.sol

Because the `AppId` for the `RelayerManager` is set in the constructor, the registration of the `RelayerManager` to the `FeeManager` can be front-run easily.

**Impact** The `AppId` registration in the `FeeManager` contract can be front-run.

**Suggestion** Revise the `AppId` registration logic.

### 2.1.3 Potential price manipulation

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the `BridgeRouter` contract, when the relayer is used, some of the user deposits are charged to pay the fees. If the bridged token is not native token, then an oracle is used to calculate the amount of tokens to be charged. As shown on Line 420 in the following code segment, the `getAmountOut` function of the `priceOracle` is used to calculate the calculate the amount of user deposits to be charged.

This function is subject to price manipulation attacks.

```
413    function relayCharge(
414        RToken.Token memory token,
415        bytes32 commitment,
416        uint256 dataLength
417    ) private returns (uint256 fee, address relayer) {
418        fee = feeManager.getFees(relayerManager.appId(), _chainId, dataLength);
419        require(fee > 0, Errors.B_ZERO_AMOUNT);
420        if (token.addr != address(0) && token.addr != address(weth9)) {
421            fee = priceOracle.getAmountOut(address(weth9), token.addr, fee);
422        }
423        relayer = relayerManager.pickRelayer(commitment);
424        require(relayer == _msgSender(), Errors.BR_WRONG_EXECUTOR);
425        token.exit(address(this), relayer, fee);
426        if (relayer.code.length > 0) {
427            bool success = IRelayerProcessor(relayer).postAcquire(
428                token.addr,
429                fee
430            );
```

```
            require(success, Errors.R_ACQUIRE_FAILED);
        }
    }
```

**Listing 2.3:** BridgeRouter.sol

Specifically, in the current implementation of the oracle, the Uniswap V2 spot price calculated by the reserve ratio is used. This ratio is well-known to be subject to price manipulation attacks. The potential loss is up to the `amount - amountMin`. If the `amountMin` is not properly set, users can lose all their deposits.

```
function getAmountOut(
    address tokenIn,
    address tokenOut,
    uint256 amountIn
) external view override returns (uint256) {
    IUniswapV2Factory factory = IUniswapV2Factory(uniRouter.factory());
    address input = (tokenIn == address(0)) ? uniRouter.WETH() : tokenIn;
    address output = (tokenOut == address(0)) ? uniRouter.WETH() : tokenOut;
    (address token0, ) = sortTokens(input, output);
    IUniswapV2Pair pair = IUniswapV2Pair(
        token0 == input
            ? factory.getPair(input, output)
            : factory.getPair(output, input)
    );
    (uint256 reserve0, uint256 reserve1, ) = pair.getReserves();
    (uint256 reserveInput, uint256 reserveOutput) = token0 == input
        ? (reserve0, reserve1)
        : (reserve1, reserve0);
    return quote(amountIn, reserveInput, reserveOutput);
}
```

**Listing 2.4:** UniswapV2DynamicPriceOracle.sol

**Impact** The gas fee calculation is subject to price manipulation when the relayer is used and fees are swapped from user deposit tokens.

**Suggestion** Fix the gas fee calculation logic.

### 2.1.4 Centralization Risk

**Severity** High

**Status** Fixed in `Version 2`

**Introduced by** `Version 1`

**Description** In the `BridgeRouter` contract, there is an `emergencyWithdraw` function which can withdraw all the balance of any token from the router to the owner. In the ExoSwap project, all users transfer their tokens to the router contract. So this function can result in severe centralization risk.

```
function emergencyWithdraw(address token) external onlyOwner {
    if (token == address(0)) {
        (bool success, ) = payable(_msgSender()).call{
            value: address(this).balance
        }("");
```

```
            require(success, Errors.B_SEND_REVERT);
        } else {
            IERC20(token).transfer(
                _msgSender(),
                IERC20(token).balanceOf(address(this))
            );
        }
    }
```

**Listing 2.5:** BridgeRouter.sol

**Impact** The `emergencyWithdraw` function can withdraw all the balance of any token from the router, which stores all user deposits.

**Suggestion** Remove this function.

## 2.2 Additional Recommendation

### 2.2.1 Require minimum number of cosigners

**Status** Acknowledged

**Introduced by** `Version 1`

**Description** In the `BridgeCosignerManager` contract, there is a minimum number (`MIN_COSIGNER_REQUIRED = 2`) of cosigners required in the `getCosignCount`.

```
    function getCosignCount(
        uint256 chainId
    ) public view override returns (uint8) {
        uint8 voteCount = (uint8(_cosaddrs[chainId].length) * 2) / 3; // 67%
        return
            MIN_COSIGNER_REQUIRED >= voteCount
                ? MIN_COSIGNER_REQUIRED
                : voteCount;
    }
```

**Listing 2.6:** BridgeCosignerManager.sol

However, in the `removeCosigner` function, it is not checked if the number of cosigners is larger than the minimum required number of cosigners.

```
    function removeCosigner(address cosaddr) public override onlyOwner {
        require(cosaddr != address(0), Errors.B_ZERO_ADDRESS);
        Cosigner memory cosigner = _cosigners[cosaddr];
        require(cosigner.active, Errors.B_ENTITY_NOT_EXIST);

        address[] storage addrs = _cosaddrs[cosigner.chainId];

        if (addrs.length > 1) {
            // move last to rm slot
            addrs[cosigner.index] = _cosaddrs[cosigner.chainId][
                addrs.length - 1
            ];
```

```
            addrs.pop();

            // change indexing
            address cosaddrLast = addrs[cosigner.index];
            _cosigners[cosaddrLast].index = cosigner.index;
        } else {
            // just remove it as 1 left
            addrs.pop();
        }

        delete _cosigners[cosaddr];

        emit CosignerRemoved(cosigner.addr, cosigner.chainId);
    }
```

**Listing 2.7:** BridgeCosignerManager.sol

**Impact** The number of cosigners can be less than `MIN_COSIGNER_REQUIRED` (2) after cosigner removal, which can lead to failure of cosigner verification.

**Suggestion** Implement checks for the minimum number of cosigners.