Security Audit Report for BridgeV2 Contracts

Date: Feb 20, 2024

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2024.03.19 17:48:13
+08'00'

Contents

1 Introduction 1

1.1 About Target Contracts 1

1.2 Disclaimer 1

1.3 Procedure of Auditing 2

1.3.1 Software Security 2

1.3.2 DeFi Security 2

1.3.3 NFT Security 2

1.3.4 Additional Recommendation 3

1.4 Security Model 3

2 Findings 4

2.1 Software Security 4

2.1.1 Missing setter for the `senderContractToEId` mapping 4

2.1.2 Lack of token address check in the `removeTokenFromWhitelist` function 5

2.1.3 Withdrawal fee is charged while not transferred to fee recipient 5

2.1.4 Incorrect order of magnitude 7

2.2 DeFi Security 8

2.2.1 Potential failed bridging due to inconsistent token addresses 8

2.3 Additional Recommendation 9

2.3.1 Remove duplicated codes 9

2.3.2 Add a check on `destChain` 10

2.3.3 Revise the duplicated handling logic in the `deposit` function 10

2.4 Notes 11

2.4.1 Accidental native token transfers are not taken into consideration 11

2.4.2 Potential centralization risks 11

2.4.3 A token cannot have both `isMinted` and `isPegged` attributes 12

2.4.4 Unverified LayerZero options 13

Report Manifest

Item	Description
Client	Spherium
Target	BridgeV2 Contracts

Version History

Version	Date	Description
1.0	Feb 20, 2024	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. 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 repository [1] for the BridgeV2 Contracts. Spherium Bridge utilizes LayerZero framework to bridge tokens from source chain to target chain.

The auditing process is iterative. Specifically, we will 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. Our audit report is responsible for the only initial version (i.e., `Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
BridgeV2 Contracts	`Version 1`	`66909f2e70d5aeaf4590361c0f0ef966c4788787`
	`Version 2`	`284bcb63a62af75503390b82c7c4e04cde9b03b8`
	`Version 3`	`5b794c89c83076bd160113c4179fdc23f7360705`
	`Version 4`	`8c9d06321e8ae9301202be91898425a5cee56f2a`
	`Version 5`	`d813960a31a3ca0b492c40cc968d641734cd19d4`
	`Version 6`	`7c2ff55df7af66b6796d88a22d1a102d8ecdc064`

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://gitlab.com/spherium/spherium-bridge/bridgev2`

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
- Access control
- 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	*High*	High	Medium
	Low	Medium	Low
		High	*Low*
		Likelihood	

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 **five** potential issues. Besides, we also have **three** recommendations and **four** notes.

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

ID	Severity	Description	Category	Status
1	High	Missing setter for the senderContractToEId mapping	Software Security	Fixed
2	Medium	Lack of token address check in the removeTokenFromWhitelist function	Software Security	Fixed
3	Low	Withdrawal fee is charged while not transferred to fee recipient	Software Security	Fixed
4	High	Incorrect order of magnitude	Software Security	Fixed
5	High	Potential failed bridging due to inconsistent token addresses	DeFi Security	Fixed
6	-	Remove duplicated codes	Recommendation	Fixed
7	-	Add a check on destChain	Recommendation	Acknowledged
8	-	Revise the duplicated handling logic in the deposit function	Recommendation	Fixed
9	-	Accidental native token transfers are not taken into consideration	Note	-
10	-	Potential centralization risks	Note	-
11	-	A token cannot have both isMinted and isPegged attributes	Note	-
12	-	Unverified LayerZero options	Note	-

The details are provided in the following sections.

2.1 Software Security

2.1.1 Missing setter for the senderContractToEId mapping

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description The BridgeReceiver contract lacks a setter function for the senderContractToEId mapping. Due to this missing functionality, the _lzReceive function will revert. With no way to add new authorized endpoints in the mapping, the contract cannot process withdrawals, rendering the bridge inoperable.

```
if (sender != senderContractToEId[senderEid]) {
    revert Sender__Not__True(0);
}
```

Listing 2.1: BridgeV2.sol

Impact The contract will be inoperable due to non-set critical variables.

Suggestion Revise the logic accordingly.

2.1.2 Lack of token address check in the `removeTokenFromWhitelist` function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `removeTokenFromWhitelist` function does not check the existence of token address to be removed from the whitelist. Although this is a privileged function, a mistaken invocation passing a non-existent address will lead to an inconsistent contract state. Specifically, it removes the first element from the `mapWhiltelistTokenNames` mapping. However, the corresponding entries in other mappings (`whitelistedTokenAddress`, `whitelistedTokenName`, `isWhitelistedAdd`, and `isWhitelistedName`) remain unchanged. This inconsistency could enable unexpected behaviors.

```
function removeTokenFromWhitelist(
    address tokenAddress
) external onlyOwner returns (bool) {
    require(tokenAddress != address(0), "Cannot be address 0");
    string memory tokenName = whitelistedTokenName[tokenAddress];
    delete whitelistedTokenAddress[tokenName];
    delete whitelistedTokenName[tokenAddress];
    uint256 i = mapWhiltelistTokenNames[tokenName];
    string memory lastTokenName = whitelistedTokenNames[
        ((whitelistedTokenNames.length) - 1)
    ];
    mapWhiltelistTokenNames[lastTokenName] = i;
    whitelistedTokenNames[i] = lastTokenName;
    whitelistedTokenNames.pop();
    delete mapWhiltelistTokenNames[tokenName];
    isWhitelistedAdd[tokenAddress] = false;
    isWhitelistedName[tokenName] = false;
    return true;
}
```

Listing 2.2: BridgeV2.sol

Impact Inconsistent variable updating may lead to unexpected behaviors.

Suggestion Revise the logic accordingly.

2.1.3 Withdrawal fee is charged while not transferred to fee recipient

Severity Low

Status Fixed in `Version 5`

Introduced by `Version 4`

Description In the `withdrawNative` function, the fee is deducted from the native tokens sent to the `_receiver`, yet these fees are not transferred to the `bridgedFeeAddress`, unlike in the `withdraw` function.

```
function withdrawNative(
    uint256 amount,
    address payable _receiver
) private returns (bool) {
    require(_receiver != address(0), "Cannot be address 0");

    require(
        isWhitelistedAdd[address(0)],
        "token not Whitelisted"
    );
    uint256 feeAmount = (amount * bridgeFeePercent);
    amount = (amount * 1000) - feeAmount;

    (bool success, ) = _receiver.call{value: amount/1000}("");
    if (success) {
        emit WITHDRAW((amount/1000), _receiver, address(0));
    } else {
        revert();
    }
    return success;
}
```

Listing 2.3: BridgeV2.sol

```
function withdraw(
    uint256 amount,
    string memory tokenName,
    address receiver
) private returns (bool) {
    address tokenAddress = whitelistedTokenAddress[tokenName];
    require(!isBlocked, "Bridge is blocked right now");
    require(
        isWhitelistedAdd[tokenAddress],
        "token not Whitelisted"
    );

    uint256 feeAmount = (amount * bridgeFeePercent);
    amount = (amount * 1000) - feeAmount;

    require(
        IERC20Mintable(tokenAddress).transfer(receiver, (amount / 1000)),
        "There was a problem transferring your tokens on destination chain"
    );
    require(
        IERC20Mintable(tokenAddress).transfer(
            bridgeFeeAddress,
            (feeAmount / 1000)
        ),
        "There was a problem transferring bridge fees to fee receiver"
    );

```

```
        emit WITHDRAW((amount / 1000), receiver, tokenAddress);
        return true;
    }
```

Listing 2.4: BridgeV2.sol

Impact The untransferred fees are locked in the contract.

Suggestion Revise the logic accordingly.

2.1.4 Incorrect order of magnitude

Severity High

Status Fixed in `Version 6`

Introduced by `Version 5`

Description In the `withdrawNative` function, the transferred fee is not handled correctly due to the incorrect order of magnitude. The correct amount should be `feeAmount / 1000`. Meanwhile, the function compares the amplified `amount` with fees already deducted against `address(this).balance` for the balance check. This is incorrect, as the raw amount without any fees deducted should be used to perform this check.

```
    function withdrawNative(
        uint256 amount,
        address payable _receiver
    ) private returns (bool success) {
        require(_receiver != address(0), "Cannot be address 0");

        require(isWhitelistedAdd[address(0)], "token not Whitelisted");
        uint256 feeAmount = (amount * bridgeFeePercent);
        amount = (amount * 1000) - feeAmount;

        if (amount > address(this).balance) {
            failedNativeTransfer[_receiver] = amount;

            emit FailedNative(_receiver, amount, block.timestamp);
        } else {
            (success, ) = _receiver.call{value: amount / 1000}("");
            (bool done, ) = bridgeFeeAddress.call{value: feeAmount}("");
            if (success && done) {
                emit WITHDRAW((amount / 1000), _receiver, address(0));
            } else {
                revert();
            }
        }
    }
```

Listing 2.5: BridgeV2.sol

Impact incorrect order of magnitude may bring unexpected behaviors.

Suggestion Revise the logic accordingly.

2.2 DeFi Security

2.2.1 Potential failed bridging due to inconsistent token addresses

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The current bridging process relies on identical token addresses on the source and destination chains. If the token addresses differ between chains, bridging that token will fail. The `BridgeV2` contract lacks a mapping for token addresses to handle such conversions. Specifically, the `deposit` function directly embeds the source chain token address into the bridging payload. This payload is then sent to the destination chain unchanged. Subsequently, the `BridgeReceiver` contract decodes the payload and calls the `withdraw` function, still passing the same token address.

```
bytes memory payload = abi.encode(tokenAddress, msg.sender, amount);

    MessagingFee memory fee = getFee(_destEid, payload, "", false);

    _lzSend(_destEid, payload, "", fee, msg.sender);
```

Listing 2.6: BridgeV2.sol

```
function _lzReceive(
    Origin calldata _origin,
    bytes32 _guid,
    bytes calldata payload,
    address, // Executor address as specified by the OApp.
    bytes calldata // Any extra data or options to trigger on receipt.
) internal override {
    // Decode the payload to get the message
    (address _token, address user, uint256 amount) = abi.decode(
        payload,
        (address, address, uint256)
    );
    // Extract the sender's EID from the origin
    uint32 senderEid = _origin.srcEid;
    bytes32 sender = _origin.sender;

    if (sender != senderContractToEId[senderEid]) {
        revert Sender__Not__True(0);
    }

    chainBridge.withdraw(amount, _token, user);

    //Emit the event

    emit MessageReceived(_token, user, amount, senderEid, sender);
}
```

Listing 2.7: BridgeV2.sol

Impact Token bridging may fail because the token addresses may differ on source and target chains.

Suggestion Revise the logic accordingly.

2.3 Additional Recommendation

2.3.1 Remove duplicated codes

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the following contracts, there are redundant logic or functions that can be removed to reduce code size and gas usage.

1. In the `BridgeV2` contract, the inheritance from `Ownable` is already declared in the parent `OAppCore` contract, and thus can be removed.
2. The `onlyOwner` modifier on the `getTokensLocked` function serves no purpose and can be removed.

```
    function getTokensLocked(
        address tokenAddress
    ) public view onlyOwner returns (uint256) {
        require(tokenAddress != address(0), "Cannot be address 0");
        return IERC20Mintable(tokenAddress).balanceOf(address(this));
    }
```

Listing 2.8: BridgeV2.sol

3. In the `withdraw` function, the check on line 427 is unnecessary and can be removed, since the function will revert the transaction on line 432 or 439 if balances are insufficient. Besides, this check is insufficient as it oversights the fee part.

```
427        require(
428            IERC20Mintable(tokenAddress).balanceOf(address(this)) >=
429                (amount / 1000),
430            "Not enough liquidity in the bridge"
431        );
432        require(
433            IERC20Mintable(tokenAddress).transfer(
434                receiver,
435                (amount / 1000)
436            ),
437            "There was a problem transferring your tokens on destination chain"
438        );
439        require(
440            IERC20Mintable(tokenAddress).transfer(
441                bridgeFeeAddress,
442                (feeAmount / 1000)
443            ),
444            "There was a problem transferring bridge fees to fee receiver"
445        );
```

Listing 2.9: BridgeV2.sol

4. In the withdraw function, multiplying and dividing the amount and fee by 1,000 are unnecessary as they fail to apply precision scaling. These duplicated arithemtic operations can be removed for gas optimization.
5. In both the BridgeReceiver and BridgeV2 contracts, the onlyOwner modifier on constructors is unnecessary and can be removed.

Impact N/A

Suggestion Remove the duplicated codes.

2.3.2 Add a check on destChain

Status Acknowledged

Introduced by Version 1

Description The deposit function emits a DEPOSIT event that includes the user-specified destChain parameter. However, there is no check that destChain matches _destEid used in the actual deposit. As a result, backends relying on the DEPOSIT event for chain resolution would receive incorrect destination chain information if destChain and _destEid differ. The DEPOSIT event could emit _destEid rather than unverified destChain, which can also mitigate this problem.

```
function deposit(
    uint32 _destEid,
    uint256 amount,
    address tokenAddress,
    string memory destChain
) external payable returns (bool) {
    ...
    _lzSend(_destEid, payload, "", fee, msg.sender);
    emit DEPOSIT((amount), msg.sender, tokenAddress, destChain);
    return true;
}
```

Listing 2.10: BridgeV2.sol

Impact N/A

Suggestion Revise the destChain check accordingly.

2.3.3 Revise the duplicated handling logic in the deposit function

Status Fixed in Version 3

Introduced by Version 1

Description The deposit function contains two conditional branches with identical logic, despite operating on different token types.

```
function deposit(
    uint32 _destEid,
    uint256 amount,
    address tokenAddress,
    string memory destChain
) external payable returns (bool) {
        ...
        else if (isMinted[tokenAddress] == true) {
            require(
                IERC20Mintable(tokenAddress).transferFrom(
                    msg.sender,
                    address(this),
                    amount
                ),
                "There was a problem transferring your tokens on source chain"
            );
        } else {
            require(
                IERC20Mintable(tokenAddress).transferFrom(
                    msg.sender,
                    address(this),
                    (amount)
                ),
                "There was a problem transferring your tokens on source chain"
            );
        }

        bytes memory payload = abi.encode(tokenAddress, msg.sender, amount);

        MessagingFee memory fee = getFee(_destEid, payload, "", false);

        _lzSend(_destEid, payload, "", fee, msg.sender);
        emit DEPOSIT((amount), msg.sender, tokenAddress, destChain);
        return true;
    }
```

Listing 2.11: BridgeV2.sol

Impact N/A

Suggestion Revise the duplicated codes accordingly.

2.4 Notes

2.4.1 Accidental native token transfers are not taken into consideration

Introduced by `Version 1`

Description The `BridgeReceiver` contract currently does not implement a method to withdraw native tokens. This poses a risk where users accidentally send native tokens to the contract and have their funds locked. The locked assets can only be withdrawn by upgrading the contract.

Feedback from the Project Users will not interact with the `BridgeReceiver`. It's only a message receiver that will get executed by the LayerZero network. Users only interact with the BridgeV2 Contract.

2.4.2 Potential centralization risks

Introduced by `Version 1`

Description The BridgeV2's owner can withdraw arbitrary tokens via the unlockToken function, which brings centralization risk here. The same concern also exists in the withdraw function, where the withdrawer is owner-approved.

```
function unlockToken(
    address tokenAddress,
    uint256 amount,
    address receiver
) external onlyOwner {
    require(tokenAddress != address(0), "Cannot be address 0");
    require(
        IERC20Mintable(tokenAddress).transfer(receiver, amount),
        "Token Unlock failed"
    );
}
```

Listing 2.12: BridgeV2.sol

```
function withdraw(
    uint256 amount,
    address tokenAddress,
    address receiver
) external onlyWithdrawer returns (bool)
```

Listing 2.13: BridgeV2.sol

Feedback from the Project This function will get handled by the governance contract which will act as the owner of the bridge.

2.4.3 A token cannot have both isMinted and isPegged attributes

Introduced by Version 1[1]

Description When isMinted and isPegged are both set to true for a single token, the deposit and withdraw functions will execute inconsistent logic. Specifically, the deposit function enters the conditional branch on line 355, which burns the deposited token to a dead address.

```
355    if (isPegged[tokenAddress] == true)
356        IERC20Mintable(tokenAddress).transferFrom(
357            msg.sender,
358            deadAddress,
359            amount
360        ); //Burn to dead address.
361    else if (isMinted[tokenAddress] == true) {
362        require(
363            IERC20Mintable(tokenAddress).transferFrom(
364                msg.sender,
365                address(this),
366                amount
367            ),
368            "There was a problem transferring your tokens on source chain"
```

[1] Fixed in Version 3

```
    );
}
```

Listing 2.14: BridgeV2.sol

However, the withdraw function enters the branch on Line 419, which transfers rather than mints tokens to the receiver address.

```
406 if (isMinted[tokenAddress] == true) {
407         require(
408             IERC20Mintable(tokenAddress).balanceOf(address(this)) >= amount,
409             "Not enough liquidity in the bridge"
410         );
411         require(
412             IERC20Mintable(tokenAddress).transfer(receiver, amount),
413             "There was a problem transferring your tokens on destination chain"
414         );
415     } else {
416         feeAmount = (amount * bridgeFeePercent);
417         amount = (amount * 1000) - feeAmount;
418
419         if (isPegged[tokenAddress] == true) {
420             IERC20Mintable(tokenAddress).mint(receiver, amount / 1000);
421             IERC20Mintable(tokenAddress).mint(
422                 bridgeFeeAddress,
423                 (feeAmount / 1000)
424             );
425         }
```

Listing 2.15: BridgeV2.sol

2.4.4 Unverified LayerZero options

Introduced by Version 2[2]

Description The deposit function in the BridgeV2 contract passes the unchecked parameter _options to _lzSend. The _options specifies _gas and _value, where _value denotes the native fee paid to Executor or other workers. Not verifying the options poses a potential risk that malicious actors could specify fees to steal funds from the BridgeV2 contract. However, since the contract is not designed to hold any native tokens, the practical impact of this risk is negligible.

```
function deposit(
    uint32 _destEid,
    uint256 amount,
    address tokenAddress,
    string memory destChain,
    bytes memory _options
) external payable returns (bool) {
    //require(tokenAddress != address(0), "Cannot be address 0");
    require(isBlocked != true, "Bridge is blocked right now");
    require(
```

[2] Fixed in Version 4

```
            isWhitelistedAdd[tokenAddress] == true,
            "This token is not Whitelisted on our platform"
        );
        require(
            amount <= IERC20Mintable(tokenAddress).balanceOf(msg.sender),
            "Amount exceeds your balance"
        );

        if (isPegged[tokenAddress] == true)
            IERC20Mintable(tokenAddress).transferFrom(
                msg.sender,
                deadAddress,
                amount
            ); //Burn to dead address.
        else if (isMinted[tokenAddress] == true) {
            require(
                IERC20Mintable(tokenAddress).transferFrom(
                    msg.sender,
                    address(this),
                    amount
                ),
                "There was a problem transferring your tokens on source chain"
            );
        } else {
            require(
                IERC20Mintable(tokenAddress).transferFrom(
                    msg.sender,
                    address(this),
                    (amount)
                ),
                "There was a problem transferring your tokens on source chain"
            );
        }

        string memory tokenName = whitelistedTokenName[tokenAddress];

        bytes memory payload = abi.encode(tokenName, msg.sender, amount);

        MessagingFee memory fee = getFee(_destEid, payload, _options, false);

        _lzSend(_destEid, payload, _options, fee, msg.sender);
        emit DEPOSIT(amount, msg.sender, tokenAddress, destChain);
        return true;
    }
```

Listing 2.16: BridgeV2.sol