Zellic

Prepared for
kled
eBridge Exchange

Prepared by
Kuilin Li
Vlad Toie
Zellic

June 5, 2024

eBridge Ethereum Bridge

Smart Contract Security Assessment

Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About eBridge Ethereum Bridge 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. MerkleTree usage complexity 11

3.2. Front-running gas attack can cancel transaction 13

3.3. Special considerations for seed liquidity 15

3.4. Manager should not be removable 18

3.5. Array-length check to be performed before adding elements 20

3.6. No way to atomically add member and requirement 23

3.7. Open receive in proxy contracts 25

3.8. Misleading colliding signatures can be crafted 26

3.9. Ownership transfer is one-step 28

3.10. Non-queued transactions can be canceled 30

4. Discussion 31

4.1. Ensure that `merkleTree` exists 32

4.2. Low daily / token bucket limits may promote front-running 32

4.3. No mechanism for retrying when BridgeOut is insolvent 33

4.4. BridgeOut must be admin in all regiments 33

5. Threat Model 34

5.1. Module: BridgeInImplementation.sol 35

5.2. Module: BridgeOutImplementationV1.sol 40

5.3. Module: BridgeOutLibrary.sol 46

5.4. Module: LimiterImplementation.sol 47

5.5. Module: MerkleTreeImplementation.sol 50

5.6. Module: RegimentImplementation.sol 54

5.7. Module: Timelock.sol 57

6. Assessment Results 62

6.1. Disclaimer 63

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for eBridge Exchange from May 14th to June 7th, 2024. During this engagement, Zellic reviewed eBridge Ethereum Bridge's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- How are the cross-chain transactions handled within the AElf ecosystem?
- Where are the actual assets stored during the cross-chain transactions?
- Which contracts handle the actual transfers of assets?
- How is the accounting performed for the incoming and outgoing assets?
- Are there any potential ways to lock the assets in the system?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

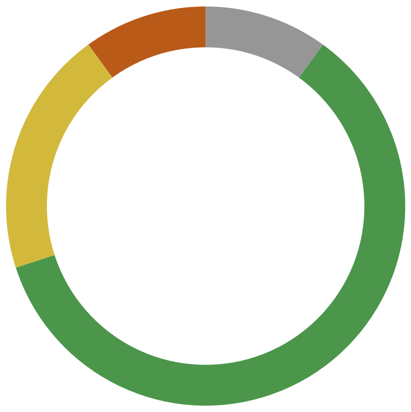
1.4. Results

During our assessment on the scoped eBridge Ethereum Bridge contracts, we discovered 10 findings. No critical issues were found. One finding was of high impact, two were of medium impact, six were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for eBridge Exchange's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	1
🟨 Medium	2
🟩 Low	6
⬜ Informational	1



2. Introduction

2.1. About eBridge Ethereum Bridge

eBridge Exchange contributed the following description of eBridge Ethereum Bridge:

> eBridge is a community-driven organization and is the first user-oriented decentralized cross-chain bridge in the AElf ecosystem.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

eBridge Ethereum Bridge Contracts

Type	Solidity
Platform	EVM-compatible

Target: ebridge-contracts.ethereum

Repository	https://github.com/eBridgeCrosschain/ebridge-contracts.ethereum ↗
Version	`c70e86a20626c02b82a7ac8189d18afaf73db173`
Programs	`contracts/`

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of five person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Kuilin Li
Engineer
kuilin@zellic.io ↗

Vlad Toie
Engineer
vlad@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 14, 2024	Kick-off call
May 14, 2024	Start of primary review period
June 7, 2024	End of primary review period

3. Detailed Findings

3.1. MerkleTree usage complexity

Target	MerkleTreeImplementation		
Category	Business Logic	**Severity**	High
Likelihood	N/A	**Impact**	High

Description

The MerkleTree is used to store cross-chain receipts of reports by a privileged actor (i.e., an oracle), through the usage of the `transmit` function.

```
function transmit(
    bytes32 swapHashId,
    bytes calldata _report,
    bytes32[] calldata _rs, // observer signatures->r
    bytes32[] calldata _ss, //observer signatures->s
    bytes32 _rawVs // signatures->v (Each 1 byte is combined into a 32-byte
    binder, which means that the maximum number of observer signatures is 32.)
) external {
    SwapInfo storage swapInfo = swapInfos[swapHashId];
    BridgeOutLibrary.ReceiptInfo memory receiptInfo = BridgeOutLibrary.Report(
        _report,
        _rs,
        _ss,
        _rawVs,
        signatureThreshold
    ).verifySignatureAndDecodeReport(swapInfo.regimentId,regiment);
    require(ledger[receiptInfo.receiptHash].leafNodeIndex == 0, "already
    recorded");
    if (receiptInfo.receiveAddress != address(0)) {
        require(receiptInfo.amount > 0, "invalid amount");
        ledger[receiptInfo.receiptHash].leafNodeIndex = 1;
        _completeReceipt(receiptInfo,swapInfo);
    }else{
        bytes32[] memory leafNodes = new bytes32[](1);
        leafNodes[0] = receiptInfo.receiptHash;
        uint256 index = IMerkleTree(merkleTree).recordMerkleTree(
            swapInfo.spaceId,
            leafNodes
        );
        ledger[receiptInfo.receiptHash].leafNodeIndex = index.add(1);
    }
```

```
        emit NewTransmission(swapHashId, msg.sender, receiptInfo.receiptId,
        receiptInfo.receiptHash);
    }
```

This in turn ensures that users can complete the receival of assets on the other side of the chain. The MerkleTree's complexity stems from the fact that all the leaves are stored on chain, which significantly deviates from the standard usage of MerkleTrees, where the leaves' path is constructed off chain and only the verification is performed on chain. This approach is not necessarily wrong, but it does introduce potential areas of concern, especially in terms of storing the leaves on chain as well as a significant gas consumption overhead on each `transmit` call.

We consider that the MerkleTree implementation is overly complex. If we are to judge the current use case, this complexity is not justifiable. We suggest storing the `receiptHash` in a mapping and redeeming it when needed, instead of storing and verifying everything as a part of a MerkleTree structure.

Impact

The complexity of the MerkleTree implementation might lead to potential issues in the future, especially when it comes to the storage of leaves on chain.

Recommendations

We recommend simplifying the MerkleTree implementation as much as possible, potentially removing the need for the MerkleTree altogether.

Remediation

This issue has been acknowledged by eBridge Exchange.

The team has additionally stated the following:

> In the transmit method, we can see two handling approaches:
> - In our previous version, we recorded the receipt information into a Merkle tree as a leaf node. Users needed to actively trigger a SwapToken transaction and verify the Merkle tree to transfer the token to themselves. (Refer to the else block, lines 5-13)
> - In our updated version, the Merkle tree will no longer be used. After validation, the token will be directly transferred to the user. (Refer to the if block, lines 1-4)
>
> We retain the original Merkle tree logic for compatibility purposes. Once we confirm that there are no compatibility issues, we will remove this part of the code.

3.2. Front-running gas attack can cancel transaction

Target	MultiSigWallet		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

When the MultiSigWallet executes a transaction, if the transaction reverts, it cannot be retried:

```
function executeTransaction(
    uint256 transactionId
) public notExecuted(transactionId) {
    if (isConfirmed(transactionId)) {
        Transaction storage transaction = transactions[transactionId];
        transaction.executed = true;
        (bool success, bytes memory returnData)
    = transaction.destination.call{
            value:  transaction.value
        }(transaction.data);
        if (success) emit Execution(transactionId);
        else {
            bytes memory returnValue = new bytes(returnData.length-4);
            for(uint i = 4; i < returnData.length;i++){
                returnValue[i-4] = returnData[i];
            }
            string memory returnValueString
    = abi.decode(returnValue,(string));
            emit ExecutionFailure(transactionId,returnValueString);
        }
    }
}
```

Note that `executed` remains true, and the current context does not revert. Also, members can confirm transactions and revoke their confirmations at any time, and the last required confirmation will execute the transaction.

Impact

Since members can revoke their confirmations at any time, any particular member who can front-run the confirmation transactions of other members can revoke right before that confirmation, ensuring that they control the transaction in which it is executed.

If a malicious member uses an out-of-gas or a call-stack attack and successfully causes the inner call to revert without reverting the outer `executeTransaction` call, then the transaction is stuck and cannot be retried.

Recommendations

We recommend unmarking the transaction as executed if the transaction reverts so that it can be retried.

Remediation

This issue has been acknowledged by eBridge Exchange, and a fix was implemented in commit `f10dd1c8` ↗.

Additionally, we note that the `ExecutionFailure` event will not be emitted, as it is no longer reachable due to the early revert.

3.3. Special considerations for seed liquidity

Target	BridgeInImplementation, BridgeOutImplementation		
Category	Code Maturity	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The team has implemented functions that specifically handle the deposit and withdrawal of seed liquidity.

```
function deposit(bytes32 tokenKey, address token, uint256 amount) external {
    require(tokenList.contains(tokenKey), "not support");
    depositAmount[tokenKey] = depositAmount[tokenKey].add(amount);
    IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    IERC20(token).safeApprove(bridgeOut, amount);
    IBridgeOut(bridgeOut).deposit(tokenKey, token, amount);
}

function withdraw(
    bytes32 tokenKey,
    address token,
    uint256 amount,
    address receiverAddress
) external onlyOwner {
    require(tokenList.contains(tokenKey), "not support");
    require(depositAmount[tokenKey] >= amount, "deposit not enough");
    depositAmount[tokenKey] = depositAmount[tokenKey].sub(amount);
    IBridgeOut(bridgeOut).withdraw(tokenKey, token, amount);
    IERC20(token).safeTransfer(receiverAddress, amount);
}
```

Essentially, the team wanted to ensure that there are enough funds for either side of the bridge to operate, under the assumption that the seed liquidity will be significant enough to cover the initial operations. As per their words,

> Initially, when no `CreateReceipt` transactions have occurred on Chain A, there are no assets in Chain A's bridgeOut contract. When a user transfers funds from another chain (Chain B) to Chain A, the initial deposit in Chain A is used to transfer funds to the user. Once there are

sufficient circulating funds within the bridge, the initial deposit can be withdrawn.

We consider that this approach is somewhat confusing, as it might not be clear for users what the intended functionalities of the `deposit` and `withdraw` function are. Technically speaking, they are not actually intended for a normal user usage, but rather they are specifically designed for provisioning of seed liquidity by the admins. This might lead to potential issues of misuse, especially since `deposit` has no restrictions on who can call it.

The same potential issues may arise due to the fact that the `deposit` function from BridgeOutImplementation has no restrictions on who can call it, and it is not clear what the intended purpose of the function is.

```
function deposit(bytes32 tokenKey, address token, uint256 amount) external {
    bytes32 swapId = tokenKeyToSwapIdMap[tokenKey];
    BridgeOutLibrary.validateToken(targetTokenList,token,tokenKey,
    swapInfos[swapId].targetToken.token);
    IERC20(token).safeTransferFrom(
        address(msg.sender),
        address(this),
        amount
    );
    tokenDepositAmount[swapId] = tokenDepositAmount[swapId].add(amount);
}
```

Impact

Users might not understand the purpose of either function and might use them for other perceived purposes, which might lead to unexpected behavior and potentially locked funds.

Recommendations

We recommend removing the `deposit` and `withdraw` functions from BridgeInImplementation and instead supplying the initial liquidity via the normal pathways that the users would use, such as `generateReceipt`. This would make the contract more straightforward and easier to understand as well as prevent potential misuse or confusion.

Alternatively, we recommend properly documenting the purpose of the `deposit` and `withdraw` functions from BridgeInImplementation as well as limiting access to the deposit functions from both the BridgeInImplementation and the BridgeOutImplementation to only the owner.

Remediation

This issue has been acknowledged by eBridge Exchange.

The team has additionally stated the following:

> Our team has proposed a version upgrade to address the issue of seed liquidity. In the next version, we will adopt the token pool method (adding and removing liquidity), remove the deposit and withdraw methods, and introduce a new contract to maintain the liquidity pool.

3.4. Manager should not be removable

Target	RegimentImplementation		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The RegimentImplementation contract is used to manage the protocol actions. It adds and removes regiment members, who can then perform particular actions across the bridges and other contracts. Each regiment also has a manager, who can add and remove members. Currently, when removing a particular member from the regiment, the manager can also be removed by mistake. This would be problematic, as no other manager can be added to the regiment, and the regiment would be left without a manager. This would prevent some specific actions from being performed, as only the manager can perform them.

```
function DeleteRegimentMember(
    bytes32 regimentId,
    address leaveMemberAddress
) external {
    RegimentInfo storage regimentInfo = regimentInfoMap[regimentId];
    EnumerableSet.AddressSet storage memberList = regimentMemberListMap[
        regimentId
    ];
    require(
        regimentInfo.manager == msg.sender,
        "Origin sender is not manager of this regiment"
    );
    require(
        memberList.contains(leaveMemberAddress),
        "member already leaved"
    );
    memberList.remove(leaveMemberAddress);
    emit RegimentMemberLeft(regimentId, leaveMemberAddress);
}
```

Impact

There are no irremediable security implications of the manager removing themselves from the regiment list, as they can just re-add themselves later. However, for consistency and posterity, it is recommended that the manager cannot be removed from the regiment.

Recommendations

We recommend ensuring that the manager cannot be removed from the regiment.

```
function DeleteRegimentMember(
    bytes32 regimentId,
    address leaveMemberAddress
) external {
    RegimentInfo storage regimentInfo = regimentInfoMap[regimentId];
    EnumerableSet.AddressSet storage memberList = regimentMemberListMap[
        regimentId
    ];
    require(
        regimentInfo.manager == msg.sender,
        "Origin sender is not manager of this regiment"
    );
    require(
        memberList.contains(leaveMemberAddress),
        "member already leaved"
    );
   require(
       regimentInfo.manager != leaveMemberAddress,
       "Manager cannot be removed from the regiment"
   );
    memberList.remove(leaveMemberAddress);
    emit RegimentMemberLeft(regimentId, leaveMemberAddress);
}
```

Remediation

This issue has been acknowledged by eBridge Exchange, and a fix was implemented in commit `f10dd1c8` ↗.

3.5. Array-length check to be performed before adding elements

Target	MerkleTreeImplementation, BridgeInImplementation		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `saveLeavesForExactTree` function in MerkleTreeImplementation allows for saving the leaf nodes of a Merkle tree. Currently, it checks whether the number of `leafs` is less than the maximum number of allowed leaves before actually adding an entire array of new leaf nodes.

```
function saveLeavesForExactTree(
    bytes32 spaceId,
    uint256 treeIndex,
    bytes32[] memory leafNodeHash,
    uint256 from,
    uint256 to
) private {
    EnumerableSet.Bytes32Set storage leafs = treeLeafList[spaceId][
        treeIndex
    ];
    require(
        leafs.length() < (1 << spaceInfoMap[spaceId].pathLength),
        "leafs exceed"
    );
    for (uint256 i = from; i <= to; i++) {
        leafs.add(leafNodeHash[i]);
    }
}
```

The `addToken` function in `BridgeIn` allows for adding new tokens to the bridge. Currently, it checks whether the number of tokens to be added is less than the maximum number of allowed tokens before actually adding the new tokens.

```
function addToken(Token[] calldata tokens) public onlyWallet {
    require(
        tokenList.length() <= MaxTokenCount && tokens.length
    <= MaxTokenCountPerAddOrRemove,
        "token count exceed"
    );
    for (uint256 i = 0; i < tokens.length; i++) {
        bytes32 tokenKey = BridgeInLibrary._generateTokenKey(
            tokens[i].tokenAddress,
            tokens[i].chainId
        );
        require(!tokenList.contains(tokenKey), "tokenKey already added");
        tokenList.add(tokenKey);
        emit TokenAdded(tokens[i].tokenAddress, tokens[i].chainId);
    }
}
```

Impact

Let us assume that the `saveLeavesForExactTree` is called. Let us say that currently, the `leafs.length() = (1 << spaceInfoMap[spaceId].pathLength) - 1`. At the same time, we can have a `leafNodehash` whose length is equal to an arbitrarily large number, say `1 << spaceInfoMap[spaceId].pathLength)`.

In this case, the `require` statement in the `saveLeavesForExactTree` function would pass, as the length of the `leafs` is less than the maximum number of allowed leafs. However, when the `leafs.add(leafNodeHash[i])` statement is executed, the `leafs` array would be filled with `leafNodeHash` elements, and the length of the `leafs` array would exceed the maximum number of allowed leaves one time over.

The same issue can be observed similarly in the `addToken`.

Recommendations

We recommend performing the length checks before adding elements to the array.

```
function saveLeavesForExactTree(
    bytes32 spaceId,
    uint256 treeIndex,
    bytes32[] memory leafNodeHash,
    uint256 from,
    uint256 to
) private {
    EnumerableSet.Bytes32Set storage leafs = treeLeafList[spaceId][
        treeIndex
    ];
    require(
        leafs.length() < (1 << spaceInfoMap[spaceId].pathLength),
        "leafs exceed"
    );
    require(
        leafs.length() + (to - from + 1) <= (1 << spaceInfoMap[spaceId].
            pathLength),
        "leafs exceed"
    );
    for (uint256 i = from; i <= to; i++) {
        leafs.add(leafNodeHash[i]);
    }
}
```

```
function addToken(Token[] calldata tokens) public onlyWallet {
    require(

        tokenList.length() <= MaxTokenCount && tokens.length <= MaxTokenCountPer
            AddOrRemove,
        "token count exceed"
    );
    require(
        tokenList.length() + tokens.length <= MaxTokenCount && tokens.length <=
            MaxTokenCountPerAddOrRemove,
        "token count exceed"
    );
    for (uint256 i = 0; i < tokens.length; i++) {
        bytes32 tokenKey = BridgeInLibrary._generateTokenKey(
            tokens[i].tokenAddress,
            tokens[i].chainId
        );
        require(!tokenList.contains(tokenKey), "tokenKey already added");
        tokenList.add(tokenKey);
        emit TokenAdded(tokens[i].tokenAddress, tokens[i].chainId);
    }
}
```

Remediation

This issue has been acknowledged by eBridge Exchange, and a fix was implemented in commit `f10dd1c8` ↗.

3.6. No way to atomically add member and requirement

Target	MultiSigWallet		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

Both adding a member and changing the requirement for the MultiSigWallet require setting up and confirming a transaction to the wallet itself to make those changes:

```
function addMember(
    address member
)
    public
    onlyWallet
    memberDoesNotExist(member)
    notNull(member)
    validRequirement(members.length.add(1), required)
{
    isMember[member] = true;
    members.push(member);
    emit MemberAddition(member);
}

function changeRequirement(
    uint256 _required
) public onlyWallet validRequirement(members.length, _required) {
    required = _required;
    emit RequirementChange(_required);
}
```

However, this means that both of these cannot be done atomically. Similarly, since removing a member is also a separate transaction, it also cannot be done in conjunction with atomically changing the requirement.

Impact

Consider a situation where the contract is a 2-of-3 multi-sig wallet. Let us assume that it is absolutely unacceptable to allow any one person to completely control the wallet. If one member wishes to rotate out their address, transferring their position in the wallet to someone else or just rotating to a new EOA private key, how can this be done safely? If the new address is added first, then that member momentarily has full control over the wallet, which is not acceptable. But, if the requirement is increased first, then the third member, who may be on vacation or may be a hard-to-access backup key, needs to be involved to confirm the addition, removal, and redecreasing of the requirement, effectively making this operation require a 3-of-3 agreement.

Recommendations

We recommend either adding the ability to sign a sequence of transactions or adding a flag to `addMember` that also adds one to the requirement value with the member, so that member exchange and similar operations do not have to temporarily change the security of the wallet.

Remediation

This issue has been acknowledged by eBridge Exchange, and a fix was implemented in commit `e45b5d20` ↗.

3.7. Open receive in proxy contracts

Target	BridgeIn, BridgeOut, Limiter, MerkleTree, Regiment		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The proxy contracts BridgeIn, BridgeOut, Limiter, MerkleTree, and Regiment all have empty payable receive functions:

```
receive() external payable {}
```

However, none of these functions except for BridgeOut actually need to receive native ETH.

Impact

Users may accidentally send native ETH to these contracts, which would lock the value until they are upgraded to allow it to be withdrawn.

Recommendations

We recommend not having a payable receive function unless the contract actually should receive transfers. In the case of WETH, the contract can whitelist the address.

Remediation

This issue has been acknowledged by eBridge Exchange, and a fix was implemented in commit `f10dd1c8` ↗.

The issue has been partly fixed in the following contracts: Limiter, MerkleTree, Regiment.

3.8. Misleading colliding signatures can be crafted

Target	Timelock		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The Timelock contract ensures that a proposed transaction is visible and auditable for a period of time after it actually executes. This adds trust to a governance process that otherwise may be completely centralized, because users who follow the Timelock contract can at least disengage with the contract under the previous implementation if they strongly disagree with the new implementation.

In addition to calldata, transactions can optionally include a `signature` field which, if present, becomes the hashed four-byte selector prepended to the calldata:

```
function executeTransaction(
    address target,
    uint value,
    string memory signature,
    bytes memory data,
    uint eta
) public payable returns (bytes memory) {
    // [...]

    bytes32 txHash = keccak256(
        abi.encode(target, value, signature, data, eta)
    );

    // [...]

    if (bytes(signature).length == 0) {
        callData = data;
    } else {
        callData = abi.encodePacked(
            bytes4(keccak256(bytes(signature))),
            data
        );
    }
```

However, the only use of this `signature` is to hash to the first four bytes. There is no requirement that it is actually the name of the function in the ABI.

Impact

A malicious caller can mislead watchers by submitting a signature that collides with the real function they are calling.

Recommendations

Since the signature itself is a property of the contract ABI, not anything actually on the blockchain, we recommend removing the `signature` field and requiring the off-chain user interface either decode or fail to decode the ABI of the called contract itself. This better represents the trust model and guarantee that the on-chain Timelock actually implements — it cannot guarantee it is calling a function with a particular signature; it can only guarantee that the first four bytes of the calldata is some data.

Remediation

This issue has been acknowledged by eBridge Exchange, and a fix was implemented in commit `f10dd1c8` ↗.

3.9. Ownership transfer is one-step

Target	RegimentImplementation		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In the RegimentImplementation, the functions `ChangeController` and `TransferRegimentOwnership` allow a previous controller and owner to transfer control and ownership, respectively, to a new address:

```
function ChangeController(
    address _controller
) external assertSenderIsController {
    require(_controller != address(0),"invalid input");
    controller = _controller;
}

function TransferRegimentOwnership(
    bytes32 regimentId,
    address newManagerAddress
) external {
    RegimentInfo storage regimentInfo = regimentInfoMap[regimentId];
    require(msg.sender == regimentInfo.manager, "no permission");
    require(newManagerAddress != address(0), "invalid manager address");
    regimentInfo.manager = newManagerAddress;
}
```

However, the new address is not checked at all, so if a user error is made, control or ownership may be lost.

Impact

User error may cause important management roles to be lost.

Recommendations

We recommend implementing a standard two-step transfer process for important roles like the controller and regiment ownership, where the previous owner nominates a next owner, and then the next owner has to accept the ownership, proving and double-checking that they can issue calls to the contract.

Remediation

This issue has been acknowledged by eBridge Exchange.

3.10. Non-queued transactions can be canceled

Target	Timelock		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the Timelock contract, the `cancelTransaction` function is used to cancel a queued transaction:

```
function cancelTransaction(
    address target,
    uint value,
    string memory signature,
    bytes memory data,
    uint eta
) public {
    require(target != address(0),"Invalid input.");
    require(
        msg.sender == admin,
        "Timelock::cancelTransaction: Call must come from admin."
    );

    bytes32 txHash = keccak256(
        abi.encode(target, value, signature, data, eta)
    );
    queuedTransactions[txHash] = false;

    emit CancelTransaction(txHash, target, value, signature, data, eta);
}
```

However, this function does not actually check if the queued transaction exists in the first place.

Impact

Transactions that were never queued and transactions that have already been executed can be canceled, which causes the emission of a misleading `CancelTransaction` event.

Recommendations

Revert if `queuedTransactions[txHash]` is not true prior to resetting it.

Remediation

This issue has been acknowledged by eBridge Exchange, and a fix was implemented in commit `f10dd1c8` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Ensure that `merkleTree` exists

In `merkleProof`, the `merkleTree` is retrieved from the `SpaceMerkleTree` mapping. It should be ensured that the `merkleTree` exists before it is used. This can be done by checking if the `merkleTreeRoot` is not equal to `bytes(0)`.

```
function merkleProof(...) external view rerturns (bool) {
    MerkleTree memory merkleTree = SpaceMerkleTree[spaceId][_treeIndex];

   require(merkleTree.merkleTreeRoot != bytes(0), "MerkleTree does not exist"
       );

    ...
}
```

A lack of such check could lead to showing that an empty `leafHash` is included in the `merkleTree` when the `merkleTree` does not exist.

4.2. Low daily / token bucket limits may promote front-running

The Limiter contract incurs transactional limits over incoming deposit requests for the BridgeIn contract as well as for the withdrawal requests of the BridgeOut contract. Currently, if the day limit is reached, either contracts will not accept any more deposits or withdrawals until the next day. This may lead to front-running attacks, where an attacker can monitor the contract and submit a transaction with a higher gas price to front-run the legitimate transaction, which will be stuck until the next day, and so on.

We recommend that the daily limit be set to a particularly high value or that the contract be designed to allow for a higher limit to be set by the owner. This would make it more difficult for an attacker to front-run the legitimate transactions.

4.3. No mechanism for retrying when BridgeOut is insolvent

The BridgeOut contract is responsible for completing the cross-chain transfer on the Ethereum side of the bridge. If the BridgeOut contract is insolvent in the token that the user desires, it will not be able to finalize the completion of the actual cross-chain transfer, even if the `report` has been marked in the MerkleTree as "completed". This could lead to a situation where the user's assets cannot be withdrawn on the destination chain, having their funds stuck in a limbo until the BridgeOut contract has enough funds to process the withdrawal.

Even though this is a highly unlikely scenario under normal circumstances, we recommend implementing a mechanism that allows the user to return their funds on the source chain if the BridgeOut contract is insolvent. This would ensure that the user's funds are not stuck in a limbo and that they can be returned to the user in case of such an event.

4.4. BridgeOut must be admin in all regiments

When a swap is being created, the BridgeOutImplementationV1 contract calls `createSpace` on the MerkleTree:

```
function createSwap(
    SwapTargetToken calldata targetToken,
    bytes32 regimentId
) external {
    // [...]

    bytes32 spaceId = IMerkleTree(merkleTree).createSpace(
        regimentId,
        defaultMerkleTreeDepth
    );
```

However, in the MerkleTreeImplementation, there is a caller check that checks that the caller is an admin of the supplied regiment:

```
function createSpace(
    bytes32 regimentId,
    uint256 pathLength
) external returns (bytes32) {
    bool isAdmin = IRegiment(regimentAddress).IsRegimentAdmin(
        regimentId,
        msg.sender
    );
    require(isAdmin, "No permission.");
```

This means that the BridgeOut contract must be an admin of all of the regiments. Since the BridgeOut address is fixed, it would be far more convenient if it were a simple check instead of having to add the same address to all the regiments.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: BridgeInImplementation.sol

Function: `createNativeTokenReceipt(string targetChainId, string targetAddress)`

Allows user to deposit native tokens into the contract for the cross-chain transfer.

Inputs

- `targetChainId`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None at this level. Checked in the `consumeReceiptLimit` function.
  - **Impact**: The target chain ID as destination for the cross-chain transfer.
- `targetAddress`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None at this level.
  - **Impact**: The target address for the funds on the target chain.

Branches and code coverage

Intended branches

- Consume the daily limit for the token by the `msg.value` for the given `targetChainId`.
  - ☑ Test coverage
- Should wrap the `msg.value` into the wrapped token. Performed via the `tokenAddress.deposit()` function.
  - ☑ Test coverage
- Should approve the `bridgeOut` contract to spend the `msg.value` amount of the wrapped token.
  - ☑ Test coverage
- Should generate a new receipt for the wrapped token.
  - ☑ Test coverage

Negative behavior

- Should not allow using more funds than `msg.value`. This is explicitly enforced through the usage of `msg.value` over additional parameters.
  - ☑ Negative test

Function: `createReceipt(address token, uint256 amount, string targetChainId, string targetAddress)`

Allows user to deposit ERC-20 tokens into the contract for the cross-chain transfer.

Inputs

- `token`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None at this level; `consumeReceiptLimit` checks if the token is supported.
  - **Impact**: The token to deposit.
- `amount`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None at this level.
  - **Impact**: The amount of tokens to deposit.
- `targetChainId`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None at this level. Checked in the `consumeReceiptLimit` function.
  - **Impact**: The target chain ID as destination for the cross-chain transfer.
- `targetAddress`
  - **Control**: Fully controlled by the user.
  - **Constraints**: None at this level.
  - **Impact**: The target address for the funds on the target chain.

Branches and code coverage

Intended branches

- Ensure that the token is supported by the bridge. Not checked at this level, but in the `consumeReceiptLimit` function.
  - ☑ Test coverage
- Transfer the `amount` of `token` from the user to the contract.
  - ☑ Test coverage
- Approve the `bridgeOut` contract to spend the `amount` of `token`.
  - ☑ Test coverage
- Generate a new receipt for the `token`.

- ☑ Test coverage

Negative behavior

- Should consume the daily limit for the token by the `amount` for the given `targetChainId`.
  - ☑ Negative test

Function: `deposit(byte[32] tokenKey, address token, uint256 amount)`

Allows the owner to deposit seed liquidity into the bridge contract.

Inputs

- `tokenKey`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be contained in the `tokenList`.
  - **Impact**: The token key to deposit.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None. Should be checked that it matches the `tokenKey`.
  - **Impact**: The token to deposit.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The amount of tokens to deposit.

Branches and code coverage

Intended branches

- Check that `token` and `tokenKey` refer to the same thing.
  - ☐ Test coverage
- Check that `tokenKey` is contained in the `tokenList`.
  - ☑ Test coverage
- Transfer the `amount` of tokens from the caller to the contract.
  - ☑ Test coverage
- Increase the `depositAmount` by the `amount` deposited.
  - ☑ Test coverage
- Deposit into the `bridgeOut` contract.
  - ☑ Test coverage
- Transfer the deposited tokens to the `bridgeOut`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner. Currently not enforced.
  - ☐ Negative test
- Should not be passed on to the other side of the chain, as the function is used for seed liquidity. Currently not explictly checked.
  - ☐ Negative test

Function: `generateReceipt(address token, uint256 amount, string targetChainId, string targetAddress)`

Generates a new receipt for the given token and amount.

Inputs

- `token`
  - **Control**: Controlled by the upper-level function.
  - **Constraints**: None at this level.
  - **Impact**: The token to generate a receipt for.
- `amount`
  - **Control**: Controlled by the upper-level function.
  - **Constraints**: None at this level.
  - **Impact**: The amount of tokens to generate a receipt for.
- `targetChainId`
  - **Control**: Controlled by the upper-level function.
  - **Constraints**: None at this level.
  - **Impact**: The target chain ID as destination for the cross-chain transfer.
- `targetAddress`
  - **Control**: Controlled by the upper-level function.
  - **Constraints**: None at this level.
  - **Impact**: The target address for the funds on the target chain.

Branches and code coverage

Intended branches

- Deposit the `amount` of `token` through the `bridgeOut` contract.
  - ☑ Test coverage
- Generate a new receipt for the `token` and store it in the `receiptIndexMap` for the given `tokenKey`.
  - ☑ Test coverage
- Increment the `totalAmountInReceipts` for the given `tokenKey`.
  - ☑ Test coverage
- Increment the `ownerToReceiptsIndexMap` for the given `msg.sender` and `tokenKey`.

  - ☑ Test coverage
- Update the `ownerToReceiptIdMap` for the given `msg.sender`, `tokenKey`, and `index`.
  - ☑ Test coverage
- Emit a `NewReceipt` event with the generated `receiptId`, `token`, `msg.sender`, and `amount`.
  - ☑ Test coverage

Negative behavior

- Should not allow overwriting existing receipts. Currently, the receipt index is incremented for each new receipt.
  - ☑ Negative test

Function: `withdraw(byte[32] tokenKey, address token, uint256 amount, address receiverAddress)`

Allows owner to withdraw the seed liquidity that has been deposited into the bridge contract.

Inputs

- `tokenKey`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be contained in the `tokenList`.
  - **Impact**: The token key to withdraw.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None. Should be checked that it matches the `tokenKey`.
  - **Impact**: The token to withdraw.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be less than the deposit amount.
  - **Impact**: The amount of tokens to withdraw.
- `receiverAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to send the withdrawn tokens to.

Branches and code coverage

Intended branches

- Check that `token` and `tokenKey` refer to the same thing. Currently not explicitly checked at this level.
  - ☐ Test coverage

- Decrease the `depositAmount` by the `amount` withdrawn.
  - ☑ Test coverage
- Withdraw from the `bridgeOut` contract.
  - ☑ Test coverage
- Transfer the withdrawn tokens to the `receiverAddress`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - ☑ Negative test
- Should not be able to withdraw more than the deposit amount.
  - ☑ Negative test
- Should not allow withdrawing unsupported tokens.
  - ☑ Negative test

5.2. Module: BridgeOutImplementationV1.sol

Function: `createSwap(SwapTargetToken targetToken, byte[32] regimentId)`

Allows creating a swap pair.

Inputs

- `targetToken`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked it does not exist in the target token list already.
  - **Impact**: The `targetToken` of the cross-chain swap.
- `regimentId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None — assumed to be valid.
  - **Impact**: The `regimentId` of the regiment that is creating the swap.

Branches and code coverage

Intended branches

- Ensure that `targetToken.token` is not the zero address.
  - ☑ Test coverage
- Ensure that the current `targetTokenList` length is less than `MaxTokenKeyCount`.
  - ☑ Test coverage
- Ensure that the `targetToken` is not already part of the target token list.
  - ☑ Test coverage

- Ensure that the `tokenKey` does not already exist.
  - ☑ Test coverage
- Create a space in the Merkle tree for the specified `regimentID`.
  - ☑ Test coverage
- Store the `swapId` information in the `swapInfos` mapping.
  - ☑ Test coverage
- Add the `tokenKey` to the `targetTokenList`.
  - ☑ Test coverage
- Map the `tokenKey` to the `swapId`.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the `regiment` manager.
  - ☑ Negative test

Function: `deposit(byte[32] tokenKey, address token, uint256 amount)`

Allows deposits for either seed liquidity or cross-chain transfer.

Inputs

- `tokenKey`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None — assumed to be valid and mapped to the `token`(checked in `validateToken`).
  - **Impact**: The `tokenKey` that is used.
- `token`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None, assumed to be valid and mapped to the `tokenKey`(checked in `validateToken`)
  - **Impact**: The `token` to be deposited.
- `amount`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None — assumed it is greater than what was deposited, via `tokenDepositAmount`.
  - **Impact**: The amount of tokens to be deposited.

Branches and code coverage

Intended branches

- Validate that the token is part of the target token list.

- ☑ Test coverage
- Transfer the token from the caller to the contract.
    - ☑ Test coverage
- Increase the token-deposit amount by the amount of the token that was deposited.
    - ☑ Test coverage

Negative behavior

- Should NOT be called by anyone other than the owner/bridgeIn contract. Currently not enforced.
    - ☐ Negative test

Function: `swapToken(byte[32] swapId, string receiptId, uint256 amount, address receiverAddress)`

Function that performs the finalization of a cross-chain transfer.

Inputs

- `swapId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Has to be a valid `swapId`.
    - **Impact**: The `swapId` is used to identify the swap info object.
- `receiptId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Has to be part of a valid leaf hash.
    - **Impact**: The `receiptId` is used to identify the receipt.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Has to be greater than zero.
    - **Impact**: The `amount` is used to determine the amount of tokens to be swapped.
- `receiverAddress`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Has to be part of a valid leaf hash.
    - **Impact**: The `receiverAddress` is used to determine the address to send the tokens to.

Branches and code coverage

Intended branches

- Ensure that a `swapInfo` exists for the specified `swapId`. Handled in `_checkParams`.
    - ☑ Test coverage

- Ensure that the `amount` is greater than zero.
  - ✓ Test coverage
- Construct the leaf hash and verify the Merkle tree.
  - ✓ Test coverage
- Ensure there is a valid `leafNodeIndex` for the `leafHash`.
  - ✓ Test coverage
- Finalize the receipt by calling `_completeReceipt`.
  - ✓ Test coverage

Negative behavior

- Should not allow calling twice. Currently not enforced. However, it is basically enforced through `tokenDepositAmount[swapId]`, which accounts for the amount of tokens deposited.
  - ☐ Negative test

Function: `transmit(byte[32] swapHashId, bytes _report, byte[32][] _rs, byte[32][] _ss, byte[32] _rawVs)`

Allows transmitting a receipt from the cross-chain transfer.

Inputs

- `swapHashId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level. Validated in `verifySignatureAndDecodeReport`.
  - **Impact**: The `swapHashId` is used to identify the swap info object.
- `_report`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level. Validated in `verifySignatureAndDecodeReport`.
  - **Impact**: The report is used to complete the receipt.
- `_rs`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level. Validated in `verifySignatureAndDecodeReport`.
  - **Impact**: The array of `r` values for the signature recovery.
- `_ss`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level. Validated in `verifySignatureAndDecodeReport`.
  - **Impact**: The arary of `r` values for the signature recovery.

- `_rawVs`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None at this level. Validated in `verifySignatureAndDecodeReport`.
    - **Impact**: The array of `v` values for the signature recovery.

Branches and code coverage

Intended branches

- Assumes that `signatureThreshold` is greater than zero. Not explicitly checked.
    - ☐ Test coverage
- Assumes no malicious intent from the behalf of the participant regiment members. Not explicitly checked.
    - ☐ Test coverage
- Assumes that the `ledger[receiptInfo.receiptHash].leafNodeIndex` is zero. Not explicitly checked.
    - ☐ Test coverage
- Verify the aggregated signatures and ensure there is enough to pass the threshold. Ensure that signatures and signers are unique and valid. Handled in `verifySignatureAndDecodeReport`.
    - ☑ Test coverage
- Ensure that `ledger[receiptInfo.receiptHash].leafNodeIndex` is still zero.
    - ☑ Test coverage
- In the case that `.receiveAddress` is not `0`, finalize the receipt (i.e., send the tokens).
    - ☑ Test coverage
- In the case that `.receiveAddress` is `0`, record the leaf node in the Merkle tree for it to be later redeemed.
    - ☑ Test coverage

Negative behavior

- The receipt should be vetted by a certain threshold of regiment membbers. Currently enforced in the `BridgeOutLibrary.Report.verifySignatureAndDecodeReport` function.
    - ☑ Negative test
- Should not be callable by anyone other than a regiment member. Handled in `verifySignatureAndDecodeReport`.
    - ☑ Negative test

Function: `withdraw(byte[32] tokenKey, address token, uint256 amount)`

Function used by the `admin` to withdraw funds that have been initially used as seed liquidity.

Inputs

- `tokenKey`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed to be valid and mapped to the `token` (checked in `validateToken`).
    - **Impact**: The `tokenKey` is used to identify the token that was initially deposited.
- `token`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed to be valid and mapped to the `tokenKey` (checked in `validateToken`).
    - **Impact**: The `token` to be withdrawn.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed it is greater than what was deposited, via `tokenDepositAmount`.
    - **Impact**: The amount of tokens to be withdrawn.

Branches and code coverage

Intended branches

- Ensure that the `tokenKey` is mapped to a valid `swapId`. Not explicitly checked but enforced through `validateToken`.
    - ☑ Test coverage
- Decrease the `tokenDepositAmount` of the `swapId` by the amount of the target token.
    - ☑ Test coverage
- Transfer the target token to the caller.
    - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the BridgeIn contract.
    - ☑ Negative test
- Assumes no malicious intent on behalf of the caller and that the admins properly manage the `bridgeIn` contract.
    - ☑ Negative test

Function: `_completeReceipt(BridgeOutLibrary.ReceiptInfo receiptInfo, SwapInfo swapInfo)`

Performs the completion of a receipt from the cross-chain transfer.

Inputs

- `receiptInfo`
    - **Control**: Controlled by the upper-level function.
    - **Constraints**: None at this level.
    - **Impact**: The receipt information is used to complete the receipt.
- `swapInfo`
    - **Control**: Controlled by the upper-level function.
    - **Constraints**: None at this level.
    - **Impact**: The swap information is used to complete the receipt.

Branches and code coverage

Intended branches

- Assumes taht `swapAmounts.receivedAmounts[token]` is 0. Not explicitly checked.
    - ☐ Test coverage
- Assure that `tokenDepostiAmount` refers to the same token that had been originally deposited. Not explicitly checked.
    - ☐ Test coverage
- Decrease the `tokenDepositAmount` of the `swapId` by the amount of the target token.
    - ☑ Test coverage
- Transfer the target token to the receiver address; if the target token is the native token, withdraw the target token amount.
    - ☑ Test coverage
- Consume the daily limit and token bucket of the swap.
    - ☑ Test coverage
- Record the received receipt.
    - ☑ Test coverage

Negative behavior

- Should not re-enter; currently, there's no explicit check against re-entry.
    - ☐ Negative test

5.3. Module: BridgeOutLibrary.sol

Function: `verifySignatureAndDecodeReport(Report report, byte[32] regimentId, address regiment)`

Function that verifies and handles the signature verification for reports.

Inputs

- `report`
    - **Control**: Fully controlled by upper-level function.
    - **Constraints**: Has to be signed by a minimum number of members of the regiment.
    - **Impact**: The report that is to be verified.
- `regimentId`
    - **Control**: Fully controlled by upper-level function.
    - **Constraints**: The caller has to be a member of the regiment.
    - **Impact**: The ID of the regiment.
- `regiment`
    - **Control**: Fully controlled by upper-level function.
    - **Constraints**: The caller has to be a member of the regiment.
    - **Impact**: The regiment that the signers are members of.

Branches and code coverage

Intended branches

- Ensure that all the signatures are valid and unique.
    - [x] Test coverage
- Ensure that all the signatures belong to regiment members.
    - [x] Test coverage
- Ensure that the count of the valid signatures goes over the threshold.
    - [x] Test coverage
- Ensure that the report is valid and that a computed hash matches the one in the report.
    - [x] Test coverage

Negative behavior

- Assumes no malicious intent on behalf of the `regimentMembers`.
    - [ ] Negative test
- Should not be callable by anyone other than `regiment` member.
    - [x] Negative test

5.4. Module: LimiterImplementation.sol

Function: `consumeDailyLimit(byte[32] dailyLimitId, address tokenAddress, uint256 amount)`

Allows consuming the daily deposit limit for a given token.

Inputs

- `dailyLimitId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints — assumed to be a valid daily limit ID.
    - **Impact**: The daily limit ID.
- `tokenAddress`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints — assumed to be a valid token address for the given daily limit ID.
    - **Impact**: The token address whose daily limit is being consumed.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints — assumed to be a valid amount.
    - **Impact**: The amount of tokens being consumed.

Branches and code coverage

Intended branches

- Update the `dailyLimit` mapping for the given `dailyLimitId` with the consumed amount.
    - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the bridge.
    - ☑ Negative test

Function: `consumeTokenBucket(byte[32] bucketId, address tokenAddress, uint256 amount)`

Allows consuming the token bucket for a given token.

Inputs

- `bucketId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints — assumed to be a valid bucket ID.
    - **Impact**: The bucket ID.
- `tokenAddress`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints — assumed to be a valid token address for the given bucket ID.

    - **Impact**: The token address whose bucket is being consumed.
- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints — assumed to be a valid amount.
    - **Impact**: The amount of tokens being consumed.

Branches and code coverage

Intended branches

- Update the `tokenBucket` mapping for the given `bucketId` with the consumed amount.
    - ☑ Test coverage

Negative behavior

- Caller is a service admin.
    - ☑ Negative test
- Negative behavior should be what the function requires.
    - ☐ Negative test

Function: `setDailyLimit(DailyLimiter.DailyLimitConfig[] dailyLimitConfigs)`

Allows setting the daily limit for a given token.

Inputs

- `dailyLimitConfigs`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints — assumed to be a valid array of daily limit configurations.
    - **Impact**: The daily limit configurations to be set.

Branches and code coverage

Intended branches

- Update the `dailyLimit` mapping for each `dailyLimitConfig` in the `dailyLimitConfigs` array.
    - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the admin.

☑ Negative test

Function: `setTokenBucketConfig(RateLimiter.TokenBucketConfig[] configs)`

Allows setting the token bucket configuration for a given token.

Inputs

- `configs`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: No constraints — assumed to be a valid array of token bucket configurations.
    - **Impact**: The token bucket configurations to be set.

Branches and code coverage

Intended branches

- Update the `tokenBucket` mapping for each `config` in the `configs` array.
    ☑ Test coverage

Negative behavior

- Should not allow anyone other than the admin to call this function.
    ☑ Negative test

5.5. Module: MerkleTreeImplementation.sol

Function: `createSpace(byte[32] regimentId, uint256 pathLength)`

Allows creating a space for a regiment.

Inputs

- `regimentId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be a valid `regimentId`.
    - **Impact**: The `regimentId` where to create the space.
- `pathLength`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must be within the range of `PathMaximalLength` and greater than zero.

- **Impact**: The path length of the space.

Branches and code coverage

Intended branches

- Ensure that the length of the `RegimentSpaceIdListMap` is less than `SpaceIdListMaximalLength`.
  - ☑ Test coverage
- Ensure that the `pathLength` is within the range of `0` and `PathMaximalLength`.
  - ☑ Test coverage
- Store the space info in the `spaceInfoMap`.
  - ☑ Test coverage
- Increment the `RegimentSpaceIndexMap`.
  - ☑ Test coverage
- Add the `spaceId` to the `RegimentSpaceIdListMap`.
  - ☑ Test coverage

Negative behavior

- No one other than a regiment admin should be able to call this function (BridgeOutImplementation).
  - ☑ Negative test

Function: `recordMerkleTree(byte[32] spaceId, byte[32][] leafNodeHash)`

Records a Merkle tree for a space.

Inputs

- `spaceId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid `spaceId`.
  - **Impact**: The `spaceId` where to record the Merkle tree.
- `leafNodeHash`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None — `msg.sender` is trusted as a regiment admin.
  - **Impact**: The `leafNodeHash` to record.

Branches and code coverage

Intended branches

- Ensure that the `spaceId` is valid.
  - ☑ Test coverage
- Save the leaves to the tree.
  - ☑ Test coverage
- Retrieve the last recorded Merkle tree index and update the Merkle tree in the case that the last recorded Merkle tree index is less than the expected index.
  - ☑ Test coverage

Negative behavior

- No one other than a regiment admin should be able to call this function (BridgeOutImplementation).
  - ☑ Negative test

Function: `saveLeavesForExactTree(byte[32] spaceId, uint256 treeIndex, byte[32][] leafNodeHash, uint256 from, uint256 to)`

Saves the leaves of a tree.

Inputs

- `spaceId`
  - **Control**: Fully controlled by upper-level functions.
  - **Constraints**: Assumed to be a valid space ID.
  - **Impact**: The space ID.
- `treeIndex`
  - **Control**: Fully controlled by upper-level functions.
  - **Constraints**: Assumed to be a valid tree index.
  - **Impact**: The tree index.
- `leafNodeHash`
  - **Control**: Fully controlled by upper-level functions.
  - **Constraints**: Assumed that the leaf node hash is unique. Not explicitly checked.
  - **Impact**: The hash of the leaf node to add to the tree.
- `from`
  - **Control**: Fully controlled by upper-level functions.
  - **Constraints**: None.
  - **Impact**: The `from` index.
- `to`
  - **Control**: Fully controlled by upper-level functions.
  - **Constraints**: None.
  - **Impact**: The `to` index.

Branches and code coverage

Intended branches

- Ensure that `leafs` are unique and have not been added before. Currently, not explicitly checked. It is checked in the `.add()` function.
  - ☐ Test coverage

Negative behavior

- Ensure that the `leafs` do not exceed the maximum leaf count. Should be checked after adding the leafs, not before.
  - ☐ Negative test

Function: `updateMerkleTree(byte[32] spaceId, uint256 treeIndex)`

Updates the Merkle tree for the given `spaceId` and `treeIndex`.

Inputs

- `spaceId`
  - **Control**: Fully controlled by the upper-level function.
  - **Constraints**: Checked that the tree attributed to it is not full.
  - **Impact**: The `spaceId` of the space info.
- `treeIndex`
  - **Control**: Fully controlled by the upper-level function.
  - **Constraints**: Checked that the tree is not full.
  - **Impact**: The index of the tree.

Branches and code coverage

Intended branches

- Ensure no `tree` exists already for the `spaceId` and `treeIndex`. Currently, not explicitly checked.
  - ☐ Test coverage
- Updates the `LastRecordedMerkleTreeIndex` to the current tree index.
  - ☑ Test coverage
- Update the `SpaceMerkleTree` to point to the new tree.
  - ☑ Test coverage
- Increments the `FullMerkleTreeCountMap` if the tree is full.
  - ☑ Test coverage
- Emits the `MerkleTreeRecorded` event.
  - ☑ Test coverage

Negative behavior

- Checks whether the tree is full or not.
  - ☑ Negative test

5.6. Module: RegimentImplementation.sol

Function: `AddRegimentMember(byte[32] regimentId, address newMemberAddress)`

Allows the addition of a new member to a regiment.

Inputs

- `regimentId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumed that `regiment` exists for the given `regimentId`.
  - **Impact**: The regiment to which the new member is to be added.
- `newMemberAddress`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should not be `address(0)`.
  - **Impact**: The new member to be added to the regiment.

Branches and code coverage

Intended branches

- Emit a `NewMemberAdded` event.
  - ☑ Test coverage
- Append `newMemberAddress` to the `memberList`.
  - ☑ Test coverage

Negative behavior

- Should not allow the addition of a member if the member limit is reached. Should be checked after adding the member.
  - ☐ Negative test
- Should not allow the addition of a member if the member is already added.
  - ☑ Negative test
- Should not allow anyone other than the manager of the regiment to call this function.
  - ☑ Negative test

Function: `CreateRegiment(address manager, address[] initialMemberList)`

Allows the creation of a regiment.

Inputs

- `manager`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should be `!= address(0)`.
  - **Impact**: The manager of the regiment.
- `initialMemberList`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should not contain `address(0)`.
  - **Impact**: The initial members of the regiment.

Branches and code coverage

Intended branches

- Assumes that the `regimentMemberListMap` is empty. Currently, no checks are performed.
  - ☐ Test coverage
- Increment `regimentCount`.
  - ☑ Test coverage
- Add `initialMemberList` to the `regimentMemberListMap`.
  - ☑ Test coverage
- If `manager` is not in `memberList`, add it.
  - ☑ Test coverage
- Ensure that the `initialMemberList` is not greater than `memberJoinLimit`.
  - ☑ Test coverage
- Set the `createTime` and `manager` in `regimentInfoMap`.
  - ☑ Test coverage
- Emit a `RegimentCreated` event.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the controller to call this function.
  - ☑ Negative test

Function: `DeleteRegimentMember(byte[32] regimentId, address leaveMemberAddress)`

Allows deletion of a member from a regiment.

Inputs

- `regimentId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed that `regiment` exists for the given `regimentId`.
    - **Impact**: The regiment from which the member is to be deleted.
- `leaveMemberAddress`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Should exist in the `memberList`.
    - **Impact**: The member to be deleted from the regiment.

Branches and code coverage

Intended branches

- Should remove the `leaveMemberAddress` from the `memberList`.
    - ☑ Test coverage
- Emit a `RegimentMemberLeft` event.
    - ☑ Test coverage

Negative behavior

- Should not allow the deletion of the manager of the regiment. Currently, no checks are performed. Alternatively, the manager should be transferred before deletion to another regiment member.
    - ☐ Negative test
- Should not allow anyone other than the manager of the regiment to call this function.
    - ☑ Negative test

Function: `TransferRegimentOwnership(byte[32] regimentId, address newManagerAddress)`

Allows performing the transfer of ownership of a regiment.

Inputs

- `regimentId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Assumed that
    - **Impact**: The regiment ID to transfer ownership of.
- `newManagerAddress`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Checked to be nonzero.
    - **Impact**: The address to transfer ownership to.

Branches and code coverage

Intended branches

- Should update the `regimentInfo.manager` with `newManagerAddress`.
  - ☑ Test coverage

Negative behavior

- Should not allow transferring the ownership to an address that is not a member of the regiment. Currently, no checks are performed.
  - ☐ Negative test
- Should not allow anyone other than the manager of the regiment to call this function.
  - ☑ Negative test

5.7. Module: Timelock.sol

Function: `acceptAdmin()`

Function that accepts the pending admin as the new admin.

Branches and code coverage

Intended branches

- Set the `admin` to `msg.sender`.
  - ☑ Test coverage
- Should reset the `pendingAdmin` to `address(0)`.
  - ☑ Test coverage
- Emit the `NewAdmin` event with the new admin as the parameter.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `pendingAdmin` to call this function.
  - ☑ Negative test

Function: `cancelTransaction(address target, uint256 value, string signature, bytes data, uint256 eta)`

Allows canceling a transaction.

Inputs

- `target`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: Cannot be the zero address.
    - **Impact**: The target address of the transaction.
- `value`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed there is a hash of the transaction it corresponds to.
    - **Impact**: The value of the transaction.
- `signature`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed there is a hash of the transaction it corresponds to.
    - **Impact**: The function signature of the transaction.
- `data`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed there is a hash of the transaction it corresponds to.
    - **Impact**: The data of the transaction.
- `eta`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed there is a hash of the transaction it corresponds to.
    - **Impact**: The estimated time of arrival of the transaction.

Branches and code coverage

Intended branches

- Calculate the transaction hash and set its value to `false` in the `queuedTransactions` mapping.
    - ☑ Test coverage
- Emit the `CancelTransaction` event with the transaction hash, target, value, signature, data, and ETA as parameters.
    - ☑ Test coverage

Negative behavior

- Ensure that the `target` is not the zero address.
    - ☑ Negative test
- Should not be callable by anyone other than the `admin`.
    - ☑ Negative test

Function: `executeTransaction(address target, uint256 value, string signature, bytes data, uint256 eta)`

Allows executing a transaction.

Inputs

- `target`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Cannot be the zero address.
    - **Impact**: The target address of the transaction.
- `value`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed there is a hash of the transaction it corresponds to.
    - **Impact**: The value of the transaction.
- `signature`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed there is a hash of the transaction it corresponds to.
    - **Impact**: The function signature of the transaction.
- `data`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed there is a hash of the transaction it corresponds to.
    - **Impact**: The data of the transaction.
- `eta`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: None — assumed there is a hash of the transaction it corresponds to.
    - **Impact**: The estimated time of arrival of the transaction.

Branches and code coverage

Intended branches

- Ensure that the transaction has been queued.
    - ☑ Test coverage
- Ensure that the transaction has surpassed the time lock.
    - ☑ Test coverage
- Ensure that the transaction is not stale.
    - ☑ Test coverage
- Set the value of the transaction hash to `false` in the `queuedTransactions` mapping.

- ☑ Test coverage
- Perform the `call` on the target address with the value and call data.
  - ☑ Test coverage
- Emit the `ExecuteTransaction` event with the transaction hash, target, value, signature, data, and ETA as parameters.
  - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `admin` to call this function.
  - ☑ Negative test

Function: `queueTransaction(address target, uint256 value, string signature, bytes data, uint256 eta)`

Allows queuing a transaction for execution.

Inputs

- `target`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Cannot be the zero address.
  - **Impact**: The target address of the transaction.
- `value`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None — assumed contract has enough balance.
  - **Impact**: The value of the transaction.
- `signature`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None at this level.
  - **Impact**: The function signature of the transaction.
- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The data of the transaction.
- `eta`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Check if the `eta` is greater than or equal to the current block timestamp plus the delay.
  - **Impact**: The estimated time of arrival of the transaction.

Branches and code coverage

Intended branches

- Check whether `eta` is greater than or equal to the current block time stamp plus the delay.
  - ☑ Test coverage
- Calculate the transaction hash and set its value to `true` in the `queuedTransactions` mapping.
  - ☑ Test coverage
- Emit the `QueueTransaction` event with the transaction hash, target, value, signature, data, and ETA as parameters.
  - ☑ Test coverage

Negative behavior

- Ensure that the `target` is not the zero address.
  - ☑ Negative test
- Should not allow anyone other than the `admin` to call this function.
  - ☑ Negative test

Function: `setDelay(uint256 delay_)`

Allows setting the delay for the time-locked actions.

Inputs

- `delay_`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than or equal to the minimum delay and less than or equal to the maximum delay.
  - **Impact**: The delay for the time-locked actions.

Branches and code coverage

Intended branches

- Should set the `delay` to the new value.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the contract itself (i.e., via callback).
  - ☑ Negative test

Function: `setPendingAdmin(address pendingAdmin_)`

Allows setting the pending admin.

Inputs

- `pendingAdmin_`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: Must not be the zero address.
    - **Impact**: The address of the pending admin.

Branches and code coverage

Intended branches

- Set the `pendingAdmin` to the new value.
    - ☑ Test coverage
- Emit the `NewPendingAdmin` event with the new pending admin as the parameter.
    - ☑ Test coverage

Negative behavior

- Should not allow anyone other than the `address(this)` to call this function.
    - ☑ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was partly deployed to the Ethereum Mainnet and the AElf Mainnet.

During our assessment on the scoped eBridge Ethereum Bridge contracts, we discovered 10 findings. No critical issues were found. One finding was of high impact, two were of medium impact, six were of low impact, and the remaining finding was informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.