BlockÂpex

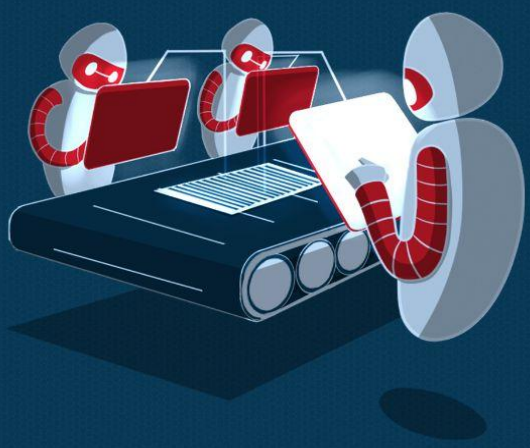
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



PREFACE

Objectives

The purpose of this document is to highlight any identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and the client's intellectual property. It also includes information on potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of the information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly to aid our growing blockchain community; at the client's discretion.

Key understandings

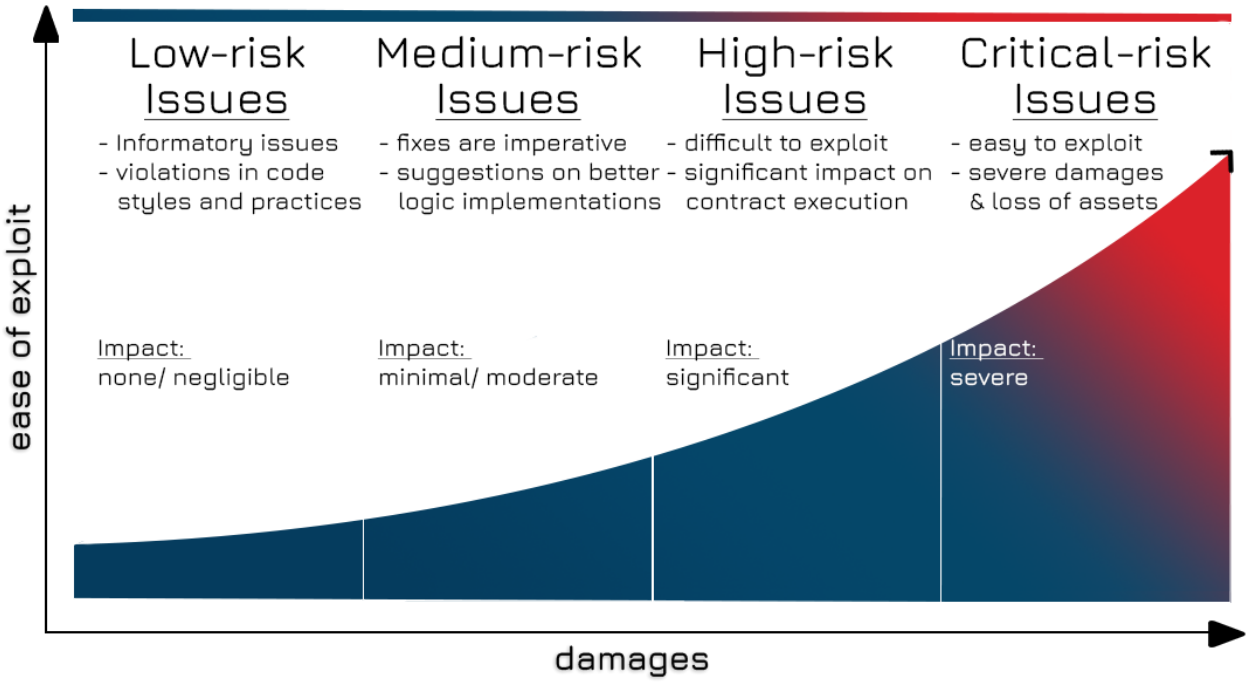


TABLE OF CONTENTS

PREFACE 2
Objectives 2
Key understandings 2
TABLE OF CONTENTS 3
INTRODUCTION 4
Scope 5
Project Overview 6
LightLink Bridge: 6
System Architecture 7
Bridge Registries: 7
Predicates Proxies: 7
Token Proxies: 8
Prerequisites: 8
Libraries: 8
Methodology & Scope 9
Focus of Security Audit 9
Security Gaps: 10
AUDIT REPORT 11
Executive Summary 11
Key Findings 12
Detailed Overview 13
High-risk issues 13
Centralization Risk Due to Majority Power Saturation 13
Medium-risk issues 15
Violation of Checks-Effects-Interactions Pattern 15
Low-risk issues 17
Missing Pause Functionality 17
Absence of Recovery Functionality 19
Missing Input Validations in Multisigable Contract 20
Length Check for Input Array Arguments 22

Informatory Issues and Optimizations 24
Lack of Events in Crucial Functions 24
Use Calldata for Function Arguments 26
Use Custom Error Messages for Revert Statements 28
Substandard Order of Layout in Registry Contracts Leading to Higher Gas Costs 30
DISCLAIMER 31

INTRODUCTION

BlockApex (Auditor) was contracted by LightLink (Client) for the purpose of conducting a Smart Contract Audit/ Code Review. This document presents the findings of our analysis, which started on 12th June '2023

Name
LightLink Bridge
Auditors
Moazzam Arif | Muhammad Jarir Uddin
Platform
Ethereum and EVM Compatible Chains | Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository/ Commit Hash
Private Repo | b8eab88ce573a7b6150c36afd62590fe21f171d0
White paper/ Documentation
Whitepaper for LightLink Network
Document log
Initial Audit Completed: June 22nd '2023
Final Audit (Fixed) Completed: July 3rd '2023

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect major issues/vulnerabilities. Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	Fungible token violations	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

LightLink Bridge is a core component of the LightLink network, a high-performance, secure, and scalable solution built on Ethereum. The bridge serves as a conduit for assets between Ethereum's Layer 1 (L1) and Layer 2 (L2), enabling fast and low-cost transactions. It is designed to unlock the power of Layer 2 scaling solutions, thereby enhancing the efficiency and scalability of the Ethereum network.

LightLink Bridge

The LightLink Bridge supports a wide range of token standards, including ERC20, ERC721, ERC1155, and native tokens, making it highly versatile and adaptable for various applications. It is built to onboard new users, offering both Enterprise mode and standard transacting. This flexibility, combined with the optionality to pay fees in either ETH or its native token, LL, makes LightLink Bridge a powerful tool for asset management on the Ethereum network.

With its highly composable nature, LightLink Bridge is suitable for a wide range of applications, including enterprise ecosystems, gaming and metaverse projects, ticketing, and identity management. It represents a significant advancement in Layer 2 scaling solutions, contributing to the ongoing development and evolution of the Ethereum ecosystem.

System Architecture

The LightLink Bridge system architecture is based on a Proof-of-Authority mechanism with an external validator set that stakes its individual reputation in order to earn incentives for the trades on the LightLink bridge. This ensures a robust and interconnected framework enabling seamless asset transfers between Ethereum's Layer 1 and Layer 2. The bridge architecture consists of several key components, each playing a crucial role in the overall operation of the bridge.

These components include Bridge Registries, Predicates Proxies, Token Proxies, and some helper libraries.

Bridge Registries:

Bridge Registries are the core of the LightLink Bridge, maintaining the mapping information for each token. The registries are divided into L1 and L2 Bridge Registries, each responsible for operations on their respective Ethereum layers.

- The L1BridgeRegistry contract manages the mapping of L1 tokens to their corresponding L2 tokens,
- while the L2BridgeRegistry contract manages the mapping of L2 tokens to their corresponding L1 tokens.

These registries ensure the accurate representation and tracking of assets across layers.

Predicates Proxies:

Token Predicates Proxies handle the deposit and withdrawal operations for different token standards. They include L1 and L2 Predicate contracts for ERC20, ERC721, ERC1155, and native tokens.

- The L1 Predicate contracts are responsible for locking &and unlocking tokens on L1 and emitting events that the L2 contracts listen for.
- Conversely, the L2 Predicate contracts listen for these events and mint & burn the corresponding tokens on L2.

These proxies ensure the secure and accurate transfer of assets between layers.

Token Proxies:

Token Proxies are responsible for creating and operating L2 tokens. They include templates for ERC20, ERC721, ERC1155, and native tokens. These templates are used to create new L2 tokens that represent the L1 assets. The Token Proxies interact with the Bridge Registries and the Predicate Proxies to ensure the accurate representation of L1 assets on L2.

Prerequisites:

The Prerequisite include Multisig contracts on both L1 and L2. These contracts provide the necessary security measures for the management of assets and the execution of transactions. The Multisig contracts require multiple signatures for certain operations, adding an extra layer of security. They are crucial for the operation of the Bridge Registries and the Predicate Proxies.

Libraries:

Libraries in the ecosystem provide additional functionalities that enhance the operation of the LightLink Bridge. They include Create2, MemberSet, and ValidatorSet libraries.

- The Create2 library is used for deterministic contract creation, specifically targeting contracts on L2 and ensuring that the same address is generated every time a contract is created with the same parameters.
- The MemberSet library manages the members of a set, providing functions for adding, removing, and checking members of the multi-sig contracts on both chains.
- The ValidatorSet library handles all the external validators in a set, providing functionality to manage the validators in order to modify the details of validators or even remove certain validators once approved by the multi-sig.

These libraries provide essential tools for the efficient operation of the LightLink Bridge.

Methodology & Scope

The codebase was audited using a filtered audit technique. A band of two (2) auditors scanned the codebase in an iterative process for a time spanning ten (10) days.

Starting with the recon phase, a basic understanding was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles, best practices, and identifying false positives detected by automated analysis tools.

Focus of Security Audit

LightLink, being a POA bridge, relies on a set of trust assumptions considering its external validators. The smart contracts' technical interpretation of this trust is reflected in the validity of off-chain signed messages, watched events, and signatures.

The focus of this audit revolved around a targeted niche of attack vectors which further expanded to a spectrum of vulnerabilities that present themselves in the scope of smart contracts and hence arise in the following cases;

- Securely managing the validators set and recording legitimate signatures.
    - Ensuring vulnerabilities relevant to signatures, e.g., Signature malleability, Signature replay, etc. are always taken care of.
- Securely managing the multi-sig members' and validators' records with the least censorship to handle malicious ones.
    - In any case of leaked keys of the above members, a pause/ revoke functionality along with recovery of funds be utilized in the LightLink Bridge smart contracts complex.
- Event emission is always up to mark.
    - In no case, the validators should receive sub-standard data to validate and verify.

Security Gaps:

Listed below is a set of security assumptions that are ensured to only work with the respective security controls if the smart contracts are implemented correctly.

1. There is no risk of complete loss if the threshold of validators' private keys is secured.
    a. We assume that all validators' private keys should be protected following the highest mode of security (e.g., hardware wallets, etc.)
    b. We assume that the threshold of validators' voting should justify the risk of a complete loss.
2. There is no risk of complete loss if the multi-sig private key is secured.
    a. We assume that the multi-sig private keys should be protected following the highest mode of security (e.g., hardware wallets, etc.)
    b. While it is a fairly low risk that all validators plus the members' private keys are compromised, the role of the multi-sig balances out the total loss.
3. The maximum impact of leak validator private keys should be calculated.
    a. Building up to the assumption in 1.a., in the worst of cases, enough validators compromised could allow attackers to perform as many withdrawals as they want.
4. Censorship is unlikely.
    a. We assume that all validators' private keys are available in case to vote out the malicious validators.

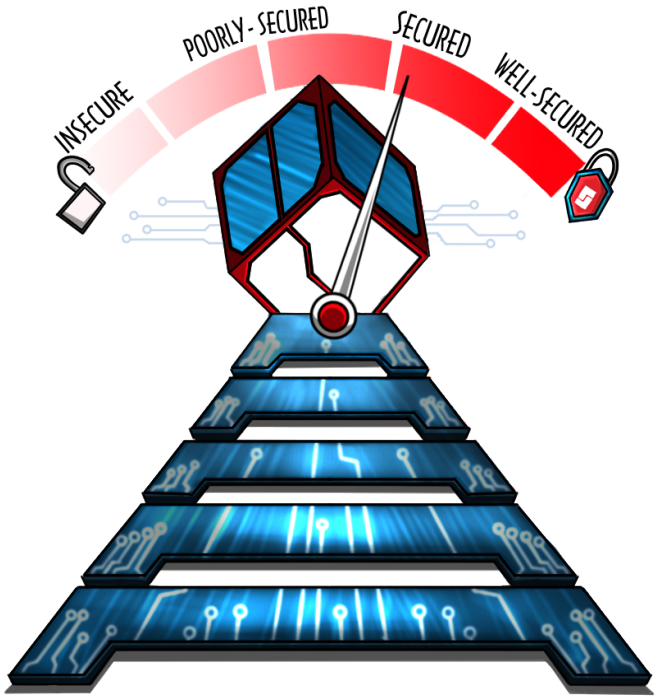
AUDIT REPORT

Executive Summary

Our team performed a technique called *Filtered Audit*, where two individuals separately audited the LightLink Bridge.

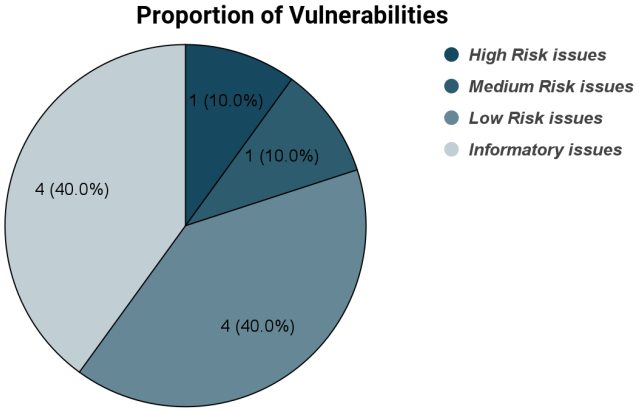
After a thorough and rigorous manual testing process involving line-by-line code review for bugs, an automated tool-based review was carried out using Slither for static analysis and Foundry for fuzzing invariants.

All the flags raised were manually reviewed and re-tested to identify the false positives.



Our team found:

#Issues	Severity Level
1	High-Risk issue(s)
1	Medium-Risk issue(s)
4	Low-Risk issue(s)
4	Informatory issue(s)



Key Findings

#	Findings	Risk	Status
1	Centralization Risk Due to Majority Power Saturation	High	Fixed
2	Violation of Checks-Effects-Interactions Pattern	Medium	Fixed
3	Missing Pause Functionality	Low	Fixed
4	Absence of Recovery Functionality	Low	Fixed
5	Missing Input Validations in Multisigable Contract	Low	Fixed
6	Length Check for Input Array Arguments	Low	Fixed
7	Lack of Events in Crucial Functions	Info	Fixed
8	Use Calldata for Function Arguments	Gas-Opt	Fixed
9	Use Custom Error Messages for Revert Statements	Gas-Opt	Resolved
10	Substandard Order of Layout Incurs Higher Gas Costs	Gas-Opt	Fixed

Detailed Overview

High-risk issues

ID	1
Title	Centralization Risk Due to Majority Power Saturation
Path	contracts/bridge/core/L1/L1BridgeRegistry.sol contracts/bridge/core/L2/L2BridgeRegistry.sol
Severity	High
Status	Fixed
Function Name	__L1BridgeRegistry_init __L2BridgeRegistry_init

Description:

The `__L1BridgeRegistry_init` and `__L2BridgeRegistry_init` functions in the respective contracts initialize two `validators` with different powers. One of the validators is assigned a majority `power` (50 out of 70), which poses a centralization risk. This `validator` has the ability to influence the rest of the `validators` and control the overall protocol direction.

Code-Affected:

The entire protocol is potentially vulnerable due to this centralization risk.

```
function __L1BridgeRegistry_init() internal {
  validators.add(0x574f879160252895a4b15fF9316bf9e9ADc46423, 50); //
Majority power
  validators.add(0x2681682d1197131D339a169dF10940470D602806, 20);
  consensusPowerThreshold = 70;
}
```

Note: The same function is initialized in the L2BridgeRegistry contract as well.

Impact:

The `validator` with an influential `powerThreshold` can control the protocol's decisions, leading to a centralized system. This is contrary to the decentralized nature of blockchain systems and could lead to misuse or manipulation of the protocol.

Recommendation:

It is recommended to distribute the power among multiple `validators` to reduce the centralization risk. This can be achieved by breaking down the `validator` power saturation into multiple `validators` right from the beginning. The updated code could look like this:

```
function __L1BridgeRegistry_init() internal {
  validators.add(0x574f879160252895a4b15fF9316bf9e9ADc46423, 35); //
Reduced power
  validators.add(0x2681682d1197131D339a169dF10940470D602806, 35); //
Increased power
  consensusPowerThreshold = 70;
}
```

Another option to prevent the `consensusPowerThreshold` from being saturated is to increase the number of `validators` in the registry contract.

Medium-risk issues

ID	2
Title	Violation of Checks-Effects-Interactions Pattern
Path	contracts/bridge/core/L1/predicates/** contracts/bridge/core/L2/predicates/**
Severity	Medium
Status	Fixed
Function Name	mapToken

Description:

The `mapToken` function in the Predicate contracts violates the Checks-Effects-Interactions pattern, potentially opening a door to reentrancy attacks. The function first interacts with the `childToken` contract by calling its `initialize` function. Only after this interaction is done, it updates its own state (`l1ToL2Gateway[l1Token]=childToken`) and emits an event (`TokenMapped`).

This coding pattern is against the best practices where during reentrancy the check for mapping `require(l1ToL2Gateway[l1Token] == _l2Token, "Token not mapped");` can be bypassed easily.

Code-Affected:

```
// call initialize using call
(bool success, bytes memory data) = childToken.call(
  abi.encodeWithSignature(
    "initialize(address,bytes)", //
    implTemplate,
    // encode function data for initialize
    abi.encodeWithSignature("initialize(address,address,address)",
multisig, address(this), l1Token)
```

```
  )
);

require(success, string(data));

// map the token
l1ToL2Gateway[l1Token] = childToken; // State changes violating CEI pattern
emit TokenMapped(messageHash);
```

Impact:

This pattern violation could potentially allow a malicious `childToken` contract to reenter the `Predicate` contract before it has a chance to update its state, leading to possible reentrancy attacks. Although in the current implementation of the smart contract complex, the pattern does not directly impact the flow of mapping tokens, in cases of malicious tokens or scenarios this practice could come in handy to an adversary.

Recommendation:

To mitigate this risk, the Predicate contracts should adhere to the Checks-Effects-Interactions pattern. This means they should perform all state changes such as `(l1ToL2Gateway[l1Token] = childToken)` before interacting with external contracts `(childToken.call())`. This, combined with a non-reentrant modifier, can eliminate the reentrancy attack window to a large extent, if not completely.

Low-risk issues

ID	3
Title	Missing Pause Functionality
Path	contracts/bridge/core/L1/predicates/** contracts/bridge/core/L2/predicates/** contracts/bridge/core/L1/L1BridgeRegistry.sol contracts/bridge/core/L2/L2BridgeRegistry.sol
Severity	Low
Status	Fixed
Function Name	-

Description:

The `BridgeRegistry` and associated `predicate` contracts do not implement a pause or `revoke` functionality. This could potentially lead to a situation where, in the event of a detected vulnerability or an ongoing attack, there is no way to immediately halt the execution of the contracts to prevent further damage or loss of funds.

Code-Affected:

The entire contract is potentially vulnerable due to the lack of a pause or revoke functionality.

Impact:

Without a `pause` or `revoke` function, the contract is vulnerable to continued exploitation in the event of a detected vulnerability or an ongoing attack. This could lead to a significant loss of funds and could potentially undermine trust in the contract.

Recommendation:

It is highly recommended to implement a pause or revoke function in the L1BridgeRegistry contract and its associated predicate contracts. This function should only be accessible

by the contract owner or a designated emergency account. This would allow for immediate action in the event of a detected vulnerability or an ongoing attack, potentially saving significant amounts of funds and preserving the integrity of the contract.

ID	4
Title	Absence of Recovery Functionality
Path	contracts/bridge/core/L1/predicates/** contracts/bridge/core/L2/predicates/** contracts/bridge/core/L1/L1BridgeRegistry.sol contracts/bridge/core/L2/L2BridgeRegistry.sol
Severity	Low
Status	Fixed
Function Name	-

Description:

The `BridgeRegistry` and associated `predicate` contracts do not implement a `recovery` functionality. This could potentially lead to a situation where, in the event of an accidental transfer or a bug causing the tokens to be stuck in a contract, there is no way to recover the funds.

Code-Affected:

The entire contract is potentially vulnerable due to the lack of a recovery functionality.

Impact:

Without a recovery function, the contract is vulnerable to permanent loss of funds in the event of an accidental transfer or a bug causing the tokens to be stuck. This could lead to significant loss of funds and could potentially undermine trust in the contract.

Recommendation:

It is recommended to implement a recovery function in the L2NativeTokenPredicate contract and its associated contracts. This function should only be accessible by the contract owner or a designated emergency account. This would allow for immediate action in the event of accidental transfers or bugs causing tokens to be stuck, potentially saving significant amounts of funds and preserving the integrity of the contract.

ID	5
Title	Missing Input Validations in Multisigable Contract
Path	contracts/bridge/prerequisite/Multisigable.sol
Severity	Low
Status	Fixed
Function Name	__Multisigable_init, modifyMultisig

Description:

The Multisigable contract does not perform input validation in the `__Multisigable_init` and `modifyMultisig` functions. Specifically, it does not check if the input address is a zero address.

Code-Affected:

```
function __Multisigable_init(address _multisig) internal {
  multisig = _multisig;
}

function modifyMultisig(address _multisig) public requireMultisig {
  multisig = _multisig;
}
```

Impact:

Without input validation, the contract is susceptible to accidental or malicious misuse. If a `zero` address is passed to these functions, it could lead to loss of control over the state variables as no valid Ethereum account corresponds to the zero address.

Recommendation:

Adding input validation checks in these functions is recommended to prevent the assignment of a zero address. This could be done by adding the following checks on top of both functions.

```
require(_multisig != address(0), "Multisig address cannot be zero")
```

ID	6
Title	Length Check for Input Array Arguments
Path	contracts/bridge/core/L1/L1BridgeRegistry.sol contracts/bridge/core/L2/L2BridgeRegistry.sol
Severity	Low
Status	Fixed
Function Name	modifyValidators, removeValidators, modifyServiceImplementations

Description:

The functions `modifyValidators`, `removeValidators`, and `modifyServiceImplementations` function in the `Registry` contracts do not perform a length check for its input array arguments. This could lead to unexpected behavior if the lengths of the `_keys` and `_implementations` arrays are not equal.

Code-Affected:

Following is one of the instances in the codebase under the scope of audit that is to be considered while applying such constraint.

```
function modifyServiceImplementations(bytes32[] memory _keys, address[]
memory _implementations) public requireMultisig {
  for (uint256 i = 0; i < _keys.length; i++) {
    implementations[_keys[i]] = _implementations[i];
  }
}
```

Impact:

If the lengths of the `_keys` and `_implementations` arrays are not equal, the function could either skip some `_keys` or attempt to access an out-of-bounds index in the `_implementations` array, leading to a revert.

Recommendation:

Add a length check at the beginning of the function to ensure that the lengths of the `_keys` and `_implementations` arrays are equal. The updated code would look like this:

```
   require(_keys.length == _implementations.length, "Input arrays must have
 the same length");
```

on both instances.

Informatory Issues and Optimizations

ID	7
Title	Lack of Events in Crucial Functions
Path	contracts/bridge/core/L1/L1BridgeRegistry.sol contracts/bridge/core/L2/L2BridgeRegistry.sol contracts/bridge/prerequisite/Multisig.sol contracts/bridge/prerequisite/Multisigable.sol
Severity	Info
Status	Fixed
Function Name	addMember, removeMember, modifyMultisig, modifyConsensusPowerThreshold, modifyValidators, removeValidators, modifyServiceImplementations

Description:

The `Registry`, `Multisig`, and `Multisigable` contracts do not emit events in the above-mentioned crucial functions;

Events are crucial in smart contracts as they provide a way to track changes and updates in the contract state. Without them, monitoring and auditing the contract's activities becomes challenging.

Code-Affected:

```
// verified
function addMember(address _account) public virtual requireMultisig {
  members.add(_account);
}
```

Impact:

The lack of events in these functions makes it difficult to track when a member is added or removed from the `multisig` and `multisigable` contracts. This could lead to unnoticed unauthorized changes in the contract's `member` set and `validators` being unaware of any ongoing attack.

Recommendation:

Emitting events in these functions is recommended to provide an audit trail of when members are `added` or `removed`. This could be done by adding emitted events at the end of each state-altering function listed above e.g. `emit MemberAdded(_account)` in the `addMember` function.

ID	8
Title	Use Calldata for Function Arguments
Path	contracts/bridge/core/L1/predicates/** contracts/bridge/core/L2/predicates/** contracts/bridge/core/L1/L1BridgeRegistry.sol contracts/bridge/core/L2/L2BridgeRegistry.sol
Severity	Gas Optimization
Status	Fixed
Function Name	All functions

Description:

The user-facing external and public functions in the `Predicate` contracts use `memory` modifiers for function arguments. However, these arguments could be declared as `calldata` to save gas, ultimately restricting any manipulation of the inputs, as `calldata` is cheaper than `memory`.

Code-Affected:

Following is one of the instances in the codebase under the scope of audit that is to be considered while applying such optimization.

```
function mapToken(address[] memory _currentValidators, bytes[] memory
_signatures, bytes memory _message) public { ... }
```

Recommendation:

Consider using `calldata` instead of `memory` for function arguments. This could save gas costs. The updated code would look like this:

```
function mapToken(address[] calldata _currentValidators, bytes[] calldata
_signatures, bytes calldata _message) public { ... }
```

ID	9
Title	Use Custom Error Messages for Revert Statements
Path	contracts/bridge/core/**
Severity	Gas Optimization
Status	Resolved
Function Name	-

Description:

Throughout the codebase under the scope of the audit, the smart contracts use require statements to enforce constraints of the system. Since the release of the Solidity 0.8.4 version, it is now supported that the require statement be translated to an if block with a custom revert error message incurring lower gas costs and improving the readability.

Code-Affected:

Following are a few of many instances in the codebase under the scope of audit that are to be considered while applying such optimization.

```
modifier requireOwner() {
  require(members.contains(msg.sender), "Owner required");
  _;
}

function add(Record storage _record, address _value) internal {
  if (contains(_record, _value)) return;
  _record.values.push(_value);
  _record.indexes[_value] = _record.values.length;
}
```

Impact:

Without a recovery function, the contract is vulnerable to permanent loss of funds in the event of an accidental transfer or a bug causing the tokens to be stuck. This could lead to a significant loss of funds and could potentially undermine trust in the contract.

Recommendation:

It is highly suggested that all the instances in the user-facing functions that are assumed to handle most of the traffic utilize the below-recommended approach which during the bull markets, the gas cost is controlled by design, and users may not need to think twice before interacting with their favorite protocols.

```
modifier requireOwner() {
  if(!members.contains(msg.sender)
     revert OwnerRequired();
  _;
}

function add(Record storage _record, address _value) internal {
  if (contains(_record, _value))
     revert RecordExists();
  _record.values.push(_value);
  _record.indexes[_value] = _record.values.length;
}
```

Developer Response:

“I believe change error format can reduce deployment cost but will not affect to function cost cause a successful transaction will not run into revert segment and of course in application level, we will not allow users to send a reverted transaction”

Auditor Response:

Acknowledged with no further action required.

ID	10
Title	Substandard Order of Layout in Registry Contracts Leading to Higher Gas Costs
Path	contracts/bridge/core/L1/L1BridgeRegistry.sol contracts/bridge/core/L2/L2BridgeRegistry.sol
Severity	Gas Optimization
Status	Fixed
Function Name	-

Description:

The current order of layout in the registry contracts does not follow the best practices suggested by the Solidity style guide. The external functions, which are frequently accessed by users, are discovered later in the bytecode traversal during each function call. This leads to higher gas costs for each transaction.

Code-Affected:

The entire contract layout is relevant to this issue. The specific code is not included here due to its size.

Impact:

The current layout order results in higher gas costs for each transaction, making the contract less efficient and more expensive for users to interact with.

Recommendation:

Following the Solidity style guide's order of layout best practices is recommended. This includes placing the contract's external functions before internal and private ones. By doing so, the external functions will be discovered earlier in the bytecode traversal, reducing the gas costs for each transaction.

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices to date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered and should not be interpreted as an influence on the potential economics of the token, its sale, or any other aspect of the project.

Crypto assets/tokens are the results of emerging blockchain technology in the domain of decentralized finance, and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service, or another asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond Solidity that could present security risks.

This audit cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.