Telepathy

Smart Contract Security Assessment

March 10, 2023

Prepared for:

Uma Roy

Succinct

Prepared by:

William Bowling and Vlad Toie

Zellic Inc.

Contents

About Zellic 3

1 Executive Summary 4

1.1 Goals of the Assessment 4

1.2 Non-goals and Limitations 4

1.3 Results 4

2 Introduction 6

2.1 About Telepathy 6

2.2 Methodology 6

2.3 Scope 8

2.4 Project Overview 8

2.5 Project Timeline 9

3 Detailed Findings 10

3.1 Missing replay protection on step 10

3.2 Frozen state not used on source chain 13

3.3 Arrays' lengths are not checked 15

3.4 Inconsistent ordering of storage variables 17

4 Discussion 19

4.1 Gas optimizations 19

4.2 Additional validation 20

4.3 Chain ID truncated to 32 bits 22

5 Threat Model 23

5.1 File: LightClient.sol 23
5.2 File: SourceAMB.sol 26
5.3 File: TargetAMB.sol 29
5.4 File: TelepathyAccess.sol 34
5.5 File: TelepathyHandlerUpgradeable.sol 37
5.6 File: TelepathyHandler.sol 39
5.7 File: TelepathyRouter.sol 40

6 Audit Results **42**

6.1 Disclaimers 42

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Succinct from February 27th to March 3rd, 2023. During this engagement, Zellic reviewed Telepathy's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 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:

- Are there any issues steming from the way contracts are upgraded?
- Is the message transmission mechanism secure? Are there ways to bypass the execution of the circuit?
- Are the proofs validated correctly on chain? Can they be used multiple times?

1.2 Non-goals and Limitations

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

- Implementation bugs in the proofs generation circuit
- Issues stemming from code or infrastructure outside of the assessment scope

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, focusing on the LightClient contract and its usage did not allow us to cover the 'libraries' component of the project as thoroughly as we covered the other components.

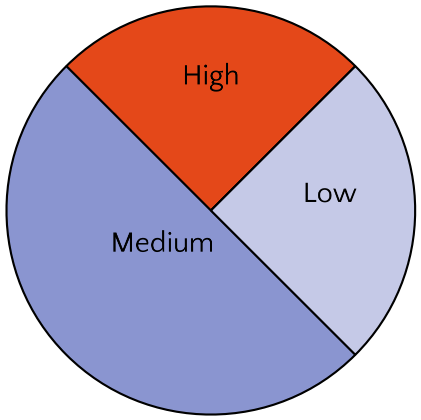
1.3 Results

During our assessment on the scoped Telepathy contracts, we discovered four findings. No critical issues were found. Of the four findings, one was of high impact, two were of medium impact, and the remaining finding was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Succinct's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	2
Low	1
Informational	0



2 Introduction

2.1 About Telepathy

Telepathy is a protocol built by Succinct Labs that allows developers to pass arbitrary messages from Ethereum to other blockchains. It uses zkSNARKs to gas-efficiently verify signatures from Ethereum validators on chain to provide information about the Ethereum state in a trust-minimized manner. It is not secured by a multisig or an optimistic fraud system, making it different from other bridging solutions. The purpose of the protocol is to implement a light client smart contract for blockchains supporting Ethereum 2 consensus and to build a cross-chain arbitrary messaging layer using the light clients as oracles.

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 the contracts' 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 code base 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, upgradeability 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.

2.3 Scope

The engagement involved a review of the following targets:

Telepathy Contracts

Repository	https://github.com/succinctlabs/telepathy-contracts
Versions	`1c747c5c7d853a9b2761062c759a5c41feebaaf4`
Programs	• TelepathyHandler • TelepathyHandlerUpgradeable • SourceAMB • TargetAMB • TelepathyAccess • TelepathyRouter • TelepathyStorage • MessageEncoding • Proxy • SimpleSerialize • StateProofHelper • Timelock • Typecast • LightClient • RotateVerifier • StepVerifier
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

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

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:

William Bowling, Engineer
vakzz@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

February 27, 2023	Kick-off call
February 27, 2022	Start of primary review period
March 3, 2023	End of primary review period

3 Detailed Findings

3.1 Missing replay protection on step

- **Target**: LightClient
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

One of the methods to update the state of the light client is the `step` function, which can be called when the light client has an existing sync committee poseidon for the requested `finalizedSlot`:

```
function step(LightClientStep memory update) external {
    bool finalized = processStep(update);

    if (getCurrentSlot() < update.attestedSlot) {
        revert("Update slot is too far in the future");
    }

    if (finalized) {
        setHead(update.finalizedSlot, update.finalizedHeaderRoot);
        setExecutionStateRoot(update.finalizedSlot,
    update.executionStateRoot);
        setTimestamp(update.finalizedSlot, block.timestamp);
    } else {
        revert("Not enough participants");
    }
}
```

If the proof is verified and finalized, the head, execution state root, and timestamp for the slot are all updated:

```
/// @notice Sets the current slot for the chain the light client is
    reflecting.
function setHead(uint256 slot, bytes32 root) internal {
    if (headers[slot] ≠ bytes32(0) && headers[slot] ≠ root) {
        consistent = false;
        return;
    }
    head = slot;
    headers[slot] = root;
    emit HeadUpdate(slot, root);
}

/// @notice Sets the execution state root for a given slot.
function setExecutionStateRoot(uint256 slot, bytes32 root) internal {
    if (executionStateRoots[slot] != bytes32(0)
    && executionStateRoots[slot] != root) {
        consistent = false;
        return;
    }
    executionStateRoots[slot] = root;
}

/// @notice Sets the sync committee poseidon for a given period.
function setSyncCommitteePoseidon(uint256 period, bytes32 poseidon)
    internal {
    if (
        syncCommitteePoseidons[period] != bytes32(0)
            && syncCommitteePoseidons[period] != poseidon
    ) {
        consistent = false;
        return;
    }
    syncCommitteePoseidons[period] = poseidon;
    emit SyncCommitteeUpdate(period, poseidon);
}

function setTimestamp(uint256 slot, uint256 timestamp) internal {
    timestamps[slot] = timestamp;
}
```

The issue is there is no check to ensure the new `finalizedSlot` is greater than the current `head` and no check to ensure that a previous call to `step` is not being replayed.

If the same `LightClientStep update` is used a second time, it will pass all of the checks and roll back the current head to the `finalizedSlot` from the previous update and set the timestamp for the slot to the current block timestamp.

Impact

As replaying a previous update will cause the timestamp for that slot to be updated, this then prevents it from being used for another five minutes due to the minimum delay:

```
/// @notice The minimum delay for using any information from the light
    client.
uint256 public constant MIN_LIGHT_CLIENT_DELAY = 60 * 5;

/// @notice Checks that the light client delay is adequate.
function requireLightClientDelay(uint64 slot, uint32 chainId)
    internal view {
    uint256 elapsedTime = block.timestamp
    - lightClients[chainId].timestamps(slot);
    require(elapsedTime ≥ MIN_LIGHT_CLIENT_DELAY, "Must wait longer to
    use this slot.");
}
```

A malicious user could continually replay an update message to prevent that slot from being used as the `requireLightClientDelay` would constantly revert.

Recommendations

A check should be added to ensure that the head slot is only ever increasing.

Remediation

The issue has been fixed in commit 485c2474.

3.2 Frozen state not used on source chain

- **Target**: SourceAMB
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Medium

Description

To send a message to another chain, a user interacts with the SourceAMB contract on the source chain, waits for the corresponding light client to be synchronized on the recipient chain, and then interacts with the TargetAMB contract on the recipient chain.

As a safety mechanism, a source chain can be frozen to prevent any messages received from being executed. The TargetAMB contract uses a `frozen` mapping to keep track of which chains are frozen. The SourceAMB contract does not use this mapping, despite inheriting the TelepathyStorage contract.

For the SourceAMB contract, a `sendingEnabled` global variable is used as an alternative, which should dictate whether or not the sending component of the messages is enabled. The naming and the states are not, however, consistent with the TargetAMB contract.

Impact

In case the TargetAMB freezes a SourceAMB chain before the `sendingEnabled` is set to `false` on the SourceAMB, the SourceAMB contract will still be able to send messages to the TargetAMB contract even though they cannot be received. This is not a security issue, but it is a potential source of confusion and could lead to unexpected behavior such as assets being locked up on one side of a bridge.

Recommendations

We recommend that the SourceAMB contract should also use the `frozen` mapping to keep track of which chains are frozen, similar to the current implementation of the TargetAMB contract.

If not already established, an off-chain rule should also be adhered to that the freezing of a chain must begin from the SourceAMB contract's side and then spread to the TargetAMB side.

This way, the freezing of a chain will be consistent across the SourceAMB and TargetAMB sides of the bridge and will be easier to reason about.

Remediation

The finding has been acknowledged by Succint. Their official response is reproduced below:

> We are not going to address this issue, as the freezing is only meant for execution side explicitly.

3.3 Arrays' lengths are not checked

- **Target**: TelepathyRouter
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Medium

Description

The `initialize` function in the TelepathyRouter contract does not check the length of the `sourceChainIds` array, which is passed as a parameter.

```
function initialize(
    uint32[] memory _sourceChainIds,
    address[] memory _lightClients,
    address[] memory _broadcasters,
    address _timelock,
    address _guardian,
    bool _sendingEnabled
) external initializer {
    // ...

    for (uint32 i = 0; i < sourceChainIds.length; i++) {
        lightClients[sourceChainIds[i]] = ILightClient(_lightClients[i]);
        broadcasters[sourceChainIds[i]] = _broadcasters[i];
        frozen[sourceChainIds[i]] = false;
    }
    sendingEnabled = _sendingEnabled;
    version = VERSION;
}
```

Impact

Due to the fact that the `initialize` function is called only once, it will likely be thoroughly tested before being deployed. However, if the `_sourceChainIds`, `_lightClients`, or `_broadcasters` arrays mismatch in length, the `initialize` function will fail and the contract will be left in a noninitialized state, allowing malicious actors that monitor the transaction pool with the possibility of calling the `initialize` function and taking control of the contract.

Recommendations

We recommend adding a check to ensure that the length of the `_sourceChainIds`, `_lightClients`, and `_broadcasters` arrays are identical.

```
function initialize(
    uint32[] memory _sourceChainIds,
    address[] memory _lightClients,
    address[] memory _broadcasters,
    address _timelock,
    address _guardian,
    bool _sendingEnabled
) external initializer {
    // ...

    require(_lightClients.length == _broadcasters.length);
    require(_lightClients.length == _sourceChainIds.length);

    for (uint32 i = 0; i < sourceChainIds.length; i++) {
        lightClients[sourceChainIds[i]] = ILightClient(_lightClients[i]);
        broadcasters[sourceChainIds[i]] = _broadcasters[i];
        frozen[sourceChainIds[i]] = false;
    }
    sendingEnabled = _sendingEnabled;
    version = VERSION;
}
```

Additionally, we recommend removing the `frozen[sourceChainIds[i]] = false;` line, as it is not needed. The `frozen` mapping is initialized to `false` by default for all keys.

Remediation

The issue has been fixed in commit 22832db0.

3.4 Inconsistent ordering of storage variables

- **Target**: TelepathyStorage
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

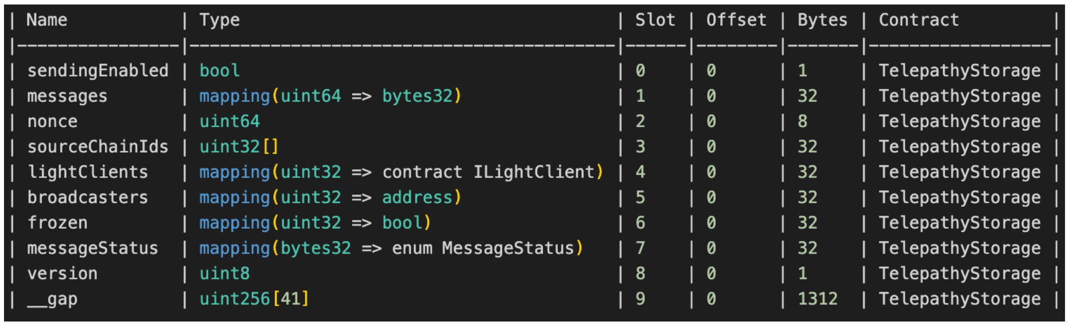
Description

Defining the `__gap` array in upgradeable contracts is a common practice to avoid the storage gap problem. However, since not all types occupy the same amount of storage slots, the `__gap` array should be ordered to maximize variable packing in order to avoid wasting storage slots.

Here is an example from the Solidity documentation.

> Ensure that you try to order your storage variables and struct members such that they can be packed tightly. For example, declaring your storage variables in the order of `uint128, uint128, uint256` instead of `uint128, uint256, uint128`, as the former will only take up two slots of storage whereas the latter will take up three.

Currently, the TelepathyStorage contract has the following storage variables:



```
| Name           | Type                                      | Slot | Offset | Bytes | Contract        |
|----------------|-------------------------------------------|------|--------|-------|-----------------|
| sendingEnabled | bool                                      | 0    | 0      | 1     | TelepathyStorage |
| messages       | mapping(uint64 => bytes32)                | 1    | 0      | 32    | TelepathyStorage |
| nonce          | uint64                                    | 2    | 0      | 8     | TelepathyStorage |
| sourceChainIds | uint32[]                                  | 3    | 0      | 32    | TelepathyStorage |
| lightClients   | mapping(uint32 => contract ILightClient)  | 4    | 0      | 32    | TelepathyStorage |
| broadcasters   | mapping(uint32 => address)                | 5    | 0      | 32    | TelepathyStorage |
| frozen         | mapping(uint32 => bool)                   | 6    | 0      | 32    | TelepathyStorage |
| messageStatus  | mapping(bytes32 => enum MessageStatus)    | 7    | 0      | 32    | TelepathyStorage |
| version        | uint8                                     | 8    | 0      | 1     | TelepathyStorage |
| __gap          | uint256[41]                               | 9    | 0      | 1312  | TelepathyStorage |
```

Figure 3.1: Storage variables initially.

The storage variables in the TelepathyStorage contract are not ordered efficiently, resulting in more storage slots being used than necessary.

Impact

The TelepathyStorage contract uses nine storage slots, while it could use only seven. As of the current implementation, this does not pose security risks, but it could become a problem in the future when the contract is upgraded and more storage variables are added.

Recommendations

We recommend sequentially reordering the variables that take less than 32 bytes and then updating the __gap array accordingly. One way to do this is shown below:

```
| Name           | Type                                     | Slot | Offset | Bytes | Contract          |
|----------------|------------------------------------------|------|--------|-------|-------------------|
| sendingEnabled | bool                                     | 0    | 0      | 1     | TelepathyStorage |
| nonce          | uint64                                   | 0    | 1      | 8     | TelepathyStorage |
| version        | uint8                                    | 0    | 9      | 1     | TelepathyStorage |
| messages       | mapping(uint64 => bytes32)               | 1    | 0      | 32    | TelepathyStorage |
| sourceChainIds | uint32[]                                 | 2    | 0      | 32    | TelepathyStorage |
| lightClients   | mapping(uint32 => contract ILightClient) | 3    | 0      | 32    | TelepathyStorage |
| broadcasters   | mapping(uint32 => address)               | 4    | 0      | 32    | TelepathyStorage |
| frozen         | mapping(uint32 => bool)                  | 5    | 0      | 32    | TelepathyStorage |
| messageStatus  | mapping(bytes32 => enum MessageStatus)   | 6    | 0      | 32    | TelepathyStorage |
| __gap          | uint256[43]                              | 7    | 0      | 1376  | TelepathyStorage |
```

Figure 3.2: Storage variables after.

Remediation

The finding has been acknowledged by Succint. Their official response is reproduced below:

> We are not going to address this issue, as we like to keep the variables grouped by source and target. We consider the gas cost of the ordered storage variables to be negligible.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Gas optimizations

In `TelepathyAccess`, the for loop can be replaced with an exists check for the mappings that are used in the function, from the current implementation:

```
// ...
    bool chainIdExists = false;
    for (uint256 i = 0; i < sourceChainIds.length; i++) {
        if (sourceChainIds[i] == chainId) {
            chainIdExists = true;
            break;
        }
    }
    if (!chainIdExists) {
        sourceChainIds.push(chainId);
        emit SourceChainAdded(chainId);
    }
     lightClients[chainId] = ILightClient(lightclient);
     broadcasters[chainId] = broadcaster;
     // ...
```

Into the following:

```
// ...
    require(chainId ≠ 0, “chainId cannot be 0”);
    require(lightclient ≠ address(0), “lightclient cannot be 0”);
    require(broadcaster ≠ address(0), “broadcaster cannot be 0”);
    if(!lightClients[chainId]) {
        sourceChainIds.push(chainId);
        emit SourceChainAdded(chainId);
    }
    lightClients[chainId] = ILightClient(lightclient);
    broadcasters[chainId] = broadcaster;
    // ...
```

4.2 Additional validation

Code maturity is very important in high-assurance projects. Checks help safeguard against unfortunate situations that might occur, help reduce the risk of lost funds and frozen protocols, and improve UX. Adding extra reverts can help clarify the internal mechanisms and reduce potential bugs that future developers might introduce while building on this project.

TargetAMB

- The `requireLightClientDelay` function checks that the light client delay is adequate. However, it does not check whether the timestamp for the slot is zero. This could happen for an invalid slot or if the slot has not been updated yet. As of the current implementation, this does not pose a security risk, as the `slot` is presumably validated in other ways in any function that calls `requireLightClientDelay`. For that reason, just adding the check for the timestamp is a good idea, as it will prevent any potential future bugs from being exploited.

```
function requireLightClientDelay(uint64 slot, uint32 chainId)
    internal view {
   require(lightClients[chainId].timestamps(slot) != 0);
    uint256 elapsedTime = block.timestamp
    - lightClients[chainId].timestamps(slot);
    require(elapsedTime >= MIN_LIGHT_CLIENT_DELAY, "Must wait longer to
    use this slot.");
}
```

- The `requireLightClientConsistency` and `requireLightClientDelay` methods both call out to the relevant light client to check the state. There is no check to ensure that `lightClients[chainId]` exists though, resulting in a revert with no message. Consider explicitly checking that the light client exists.

```
/// @notice Checks that the light client for a given chainId is consistent.
function requireLightClientConsistency(uint32 chainId) internal view {
    require(lightClients[chainId].consistent(), "Light client is
    inconsistent.");
}

/// @notice Checks that the light client delay is adequate.
function requireLightClientDelay(uint64 slot, uint32 chainId)
    internal view {
    uint256 elapsedTime = block.timestamp
    - lightClients[chainId].timestamps(slot);
    require(elapsedTime ≥ MIN_LIGHT_CLIENT_DELAY, "Must wait longer to
    use this slot.");
}
```

SourceAMB

- The `send` methods lack a check to ensure that `recipientChainId` is not `block.chainid`. Despite there being a check on the `TargetAMB` side against this particular issue, it is good practice to add a check on the `SourceAMB` side as well. This will prevent any potential future bugs on this matter.

```
function send(uint32 recipientChainId, bytes32 recipientAddress,
    bytes calldata data)
    external
    isSendingEnabled
    returns (bytes32)
{
    require(recipientChainId ≠ block.chainid);
//  ...
}

function send(uint32 recipientChainId, address recipientAddress,
    bytes calldata data)
    external
    isSendingEnabled
    returns (bytes32)
{
    require(recipientChainId ≠ block.chainid);
//  ...
}
```

4.3 Chain ID truncated to 32 bits

When the chain ID is used as a method argument or stored in TelepathyStorage, it is currently truncated to a `uint32` value from a `uint256`. The majority of chains use a value that is smaller than 32 bits, but there is technically nothing stopping one from having a chain ID larger than that. These chains would currently not be able use the TelepathyRouter as the check `message.recipientChainId` $\neq$ `block.chainid` would always fail.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart 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 File: LightClient.sol

Function: `rotate(LightClientRotate update)`

Performs a rotation of the sync committee, setting the next sync committee and approving the next update slot.

Inputs

- `update`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The constraints are defined in the `processStep` function. Should include assuring that the `attestedSlot` is not too far in the future nor in the past, similar to the `step` function.
  - **Impact**: Contains the parameters for the rotation, including the new sync committee, the new update slot, and the proof of the rotation.

Branches and code coverage (including function calls)

Intended branches

- Assumes that the update's step is the current step.
  ☐ Test coverage
- Assumes the update is legitimate and that its proof is correct.
  ☐ Test coverage
- Should update the sync committee for the next period.
  ☑ Test coverage

Negative behavior

- Should revert if the update is not legitimate in terms of the proof.

  - ☑ Negative test
- Should not allow performing an update on previous slots.
  - ☐ Negative test
- Should not be callable if not enough signatures are provided.
  - ☑ Negative test

Function call analysis

- `processStep(update.step)`

- **What is controllable?** `update.step`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value tells whether the `update.participation` is greater than the finality threshold. If it is not, the update has not been finalized and the function reverts.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Means that the previous update has not been finalized, and the function reverts.
- `setSyncCommitteePoseidon(nextPeriod, update.syncCommitteePoseidon)`
  - **What is controllable?** `update.syncCommitteePoseidon`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts it means that the update is wrong / not legitimate. It is assumed that most of the checks have been done before this point.

Function: `step(LightClientStep update)`

Does a light client update, changing the head of the light client to the provided update slot.

Inputs

- `update`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: The constraints are defined in the `processStep` function. Should include assuring that the `attestedSlot` is not too far in the future nor in the past.
  - **Impact**: The head of the light client is updated to the provided slot.

Branches and code coverage (including function calls)

Intended branches

- Should update the head.
  - ☑ Test coverage
- Should update the execution state root.
  - ☑ Test coverage
- Should update the timestamp.
  - ☑ Test coverage
- Should revert if the update slot is too far in the future.
  - ☑ Test coverage
- Should revert if the update slot is in the past. Currently not performed.
  - ☐ Test coverage

Negative behavior

- Should revert if the update is not legitimate in terms of the sync committee.
  - ☑ Negative test
- Should revert if the update is not legitimate in terms of the finality proof.
  - ☑ Negative test
- Should not allow performing an update on previous slots.
  - ☐ Negative test
- Should not be callable if not enough signatures are provided.
  - ☑ Negative test
- Should not allow performing an update on a step that was not approved by the sync committee.
  - ☑ Negative test

Function call analysis

- `processStep(update)`
- **What is controllable?**: `update`.
  - **If return value controllable, how is it used and how can it go wrong?**: Return value tells whether the `update.participation` is greater than the finality threshold. If it is not, the update has not been finalized and the function reverts.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Means that the update is wrong if it reverts.
- `zkLightClientStep(update)`
- **What is controllable?**: `update`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If it reverts, it means that the proof has not been verified, and the update

is wrong / not legitimate.

- `setHead(update.finalizedSlot, update.finalizedHeaderRoot)`

- **What is controllable?**: `update`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If it reverts, it means that the update is wrong / not legitimate. It is assumed that most of the checks have been done before this point.
- `setExecutionStateRoot(update.finalizedSlot, update.executionStateRoot)`

- **What is controllable?** `update`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If it reverts, it means that the update is wrong / not legitimate. It is assumed that most of the checks have been done before this point.
- `setTimestamp(update.finalizedSlot, update.timestamp)`
  - **What is controllable?** `update`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** No checks are done here, so it should not revert.

5.2 File: SourceAMB.sol

Function: `sendViaStorage(uint32 recipientChainId, address recipientAddress, byte[] data)`

Sends a message to the target chain by saving the `messageRoot` in the storage variable `messages` as well as emitting the `SentMessage` event. This will be picked up by the corresponding light client and can then be executed on the target chain either via `executeMessage` or `executeMessageFromLog`.

Inputs

- `recipientChainId`
  - **Control**: Full control.
  - **Constraints**: The `block.chainid` of the TargetAMB contract on the target chain executing this message must match this value.
  - **Impact**: N/A.
- `recipientAddress`
  - **Control**: Full control.
  - **Constraints**: The address must implement `ITelepathyHandler`.

    - **Impact**: When executed on the target chain, `handleTelepathy` will be called on this address.
- `data`
    - **Control**: Full control.
    - **Constraints**: No checks.
    - **Impact**: When executed on the target chain, `handleTelepathy` will be called with this data.

Branches and code coverage (including function calls)

Intended branches

- The `messageRoot` is stored against the current nonce, and the `SentMessage` event is emitted with the current nonce, the hash of the message, and the message itself.
    - ☐ Test coverage

Function: `sendViaStorage(uint32 recipientChainId, byte[32] recipientAddress, byte[] data)`

Sends a message to the target chain by saving the `messageRoot` in the storage variable `messages` as well as emitting the `SentMessage` event. This will be picked up by the corresponding light client and can then be executed on the target chain either via `executeMessage` or `executeMessageFromLog`.

Inputs

- `recipientChainId`
    - **Control**: Full control.
    - **Constraints**: The `block.chainid` of the TargetAMB contract on the target chain executing this message must match this value.
    - **Impact**: N/A.
- `recipientAddress`
    - **Control**: Full control.
    - **Constraints**: The address must implement `ITelepathyHandler`.
    - **Impact**: When executed on the target chain, `handleTelepathy` will be called on this address.
- `data`
    - **Control**: Full control.
    - **Constraints**: No checks.
    - **Impact**: When executed on the target chain, `handleTelepathy` will be called with this data.

Branches and code coverage (including function calls)

Intended branches

- The `messageRoot` is stored against the current nonce, and the `SentMessage` event is emitted with the current nonce, the hash of the message, and the message itself.
  - ☐ Test coverage

Function: `send(uint32 recipientChainId, byte[32] recipientAddress, byte[] data)`

Sends a message to the target chain by emitting the `SentMessage` event. This will be picked up by the corresponding light client and can then be executed on the target chain.

Inputs

- `recipientChainId`
  - **Control**: Full control.
  - **Constraints**: The `block.chainid` of the TargetAMB contract on the target chain executing this message must match this value.
  - **Impact**: N/A.
- `recipientAddress`
  - **Control**: Full control.
  - **Constraints**: The address must implement `ITelepathyHandler`.
  - **Impact**: When executed on the target chain, `handleTelepathy` will be called on this address.
- `data`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: When executed on the target chain, `handleTelepathy` will be called with this data.

Branches and code coverage (including function calls)

Intended branches

- A `SentMessage` event is emitted with the current nonce, the hash of the message, and the message itself.
  - ☐ Test coverage

Function: `send(uint32 recipientChainId, address recipientAddress, byte[] data)`

Sends a message to the target chain by emitting the `SentMessage` event. This will be picked up by the corresponding light client and can then be executed on the target chain.

Inputs

- `recipientChainId`
  - **Control**: Full control.
  - **Constraints**: The `block.chainid` of the TargetAMB contract on the target chain executing this message must match this value.
  - **Impact**: N/A.
- `recipientAddress`
  - **Control**: Full control.
  - **Constraints**: The address must implement `ITelepathyHandler`.
  - **Impact**: When executed on the target chain, `handleTelepathy` will be called on this address.
- `data`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: When executed on the target chain, `handleTelepathy` will be called with this data.

Branches and code coverage (including function calls)

Intended branches

- A `SentMessage` event is emitted with the current nonce, the hash of the message, and the message itself.
  ☑ Test coverage

5.3 File: TargetAMB.sol

Function: `executeMessageFromLog(byte[] srcSlotTxSlotPack, byte[] messageBytes, byte[32] receiptsRootProof, byte[32] receiptsRoot, byte[] receiptProof, byte[] txIndexRLPEncoded, uint256 logIndex)`

Executes a message given an event proof.

Inputs

- `srcSlotTxSlotPack`
  - **Control**: Fully controlled by the user.
  - **Constraints**: No direct constraints.
  - **Impact**: The slot where we want to read the header from and the slot where the tx executed, packed as two uint64s.
- `messageBytes`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Enforced in `_checkPreconditions`. It's checked that the messageStatus is `NOT_EXECUTED`, that it's chainId is the same as the current chain, and that the version is the same as the current version.
  - **Impact**: The message will be executed.
- `receiptsRootProof`
  - **Control**: Fully controlled by the user.
  - **Constraints**: No direct constraints. Should be checked in`verifyReceiptsRoot`.
  - **Impact**: Merkle proof proving the receiptsRoot in the block header.
- `receiptsRoot`
  - **Control**: Fully controlled by the user.
  - **Constraints**: No direct constraints. Should be checked in`verifyReceiptsRoot`.
  - **Impact**: The receipts root which contains the "SentMessage" event.
- `receiptProof`
  - **Control**: Fully controlled by the user.
  - **Constraints**: No direct constraints.
  - **Impact**: The receipt's proof used for verifying the "SentMessage" event.
- `txIndexRLPEncoded`
  - **Control**: Fully controlled by the user.
  - **Constraints**: No direct constraints.
  - **Impact**: The index of the transaction in the block RLP encoded.
- `logIndex`
  - **Control**: Fully controlled by the user.
  - **Constraints**: No direct constraints.
  - **Impact**: The index of the event in the transaction.

Branches and code coverage (including function calls)

Intended branches

- The light client must be consistent for that source chain.
  - [x] Test coverage
- The light client must not be frozen for that source chain.
  - [x] Test coverage
- The message must not be too old.
  - [ ] Test coverage
- The message must not have been executed before.
  - [ ] Test coverage
- The message must be valid.
  - [ ] Test coverage
- Message must be executed successfully, via `encodeWithSelector`. Any failure should be stored in `messages`.
  - [x] Test coverage
- Assumes that the `lightClients[chainId].timestamps(slot)` is not 0. Currently, this is not enforced.
  - [ ] Test coverage
- Assure that `storageRoot` is not 0. Currently, this is not enforced.
  - [ ] Test coverage
- Assure that `executionStateRoot` is not 0. Currently, this is not enforced.
  - [ ] Test coverage
- Ensure that the verified `receiptMessageRoot` matches the `messageRoot`, and that neither are 0.
  - [x] Test coverage
- Assure that the proofs are valid and that none can be used more than once.
  - [ ] Test coverage

Negative behaviour

- Should not allow re-using previous proofs of any kind.
  - [ ] Negative test

Function call analysis

- `SSZ.verifyReceiptsRoot(receiptsRoot, receiptsRootProof, headerRoot, srcSlot, txSlot)`
  - **What is controllable?**: The `receiptsRoot` and `receiptsRootProof` are controllable by the caller.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is used to verify that the receipts root is correct. If it is incorrect, the function returns false.
  - **What happens if it reverts, reenters, or does other unusual control flow?**:

n/a

- `recipient.call(receiveCall);`
  - **What is controllable?**: `receiveCall` is partly controlled by the user.
  - **If return value controllable, how is it used and how can it go wrong?**: n/a. The constraint here is that the `recipient` must implement the `ITelepathyHandler` interface. That's enforced through constructing the `receiveCall` variable with the selector of the `ITelepathyHandler.handleTelepathy` function.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Arbitrary code execution on the target `recipient` contract. Should be safe, as the `recipient` contract must implement the `ITelepathyHandler` interface.

Function: `executeMessage(uint64 slot, byte[] messageBytes, byte[] accountProof, byte[] storageProof)`

Allows executing a message given a storage proof.

Inputs

- `slot`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Must be a valid slot. Its light client's timestamp must be within the last 5 minutes.
  - **Impact**: The slot will be used to get the execution state root from the light client.
- `messageBytes`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Enforced in `_checkPreconditions`. It is checked that the messageStatus is `NOT_EXECUTED`, that its `chainId` is the same as the current chain, and that the version is the same as the current version.
  - **Impact**: The message will be executed.
- `accountProof`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Assumed a valid account proof.
  - **Impact**: The account proof will be used to prove the broadcaster's state root.
- `storageProof`
  - **Control**: Fully controlled by the user.
  - **Constraints**: Assumed a valid storage proof.
  - **Impact**: The storage proof will be used to prove the existence of the message root inside the broadcaster.

Branches and code coverage (including function calls)

Intended branches

- The light client must be consistent for that source chain.
  - [x] Test coverage
- The light client must not be frozen for that source chain.
  - [x] Test coverage
- The message must not be too old.
  - [ ] Test coverage
- The message must not have been executed before.
  - [ ] Test coverage
- The message must be valid.
  - [ ] Test coverage
- Message must be executed successfully via `encodeWithSelector`. Any failure should be stored in `messages`.
  - [x] Test coverage
- Assumes that the `lightClients[chainId].timestamps(slot)` is not 0. Currently, this is not enforced.
  - [ ] Test coverage
- Assure that `storageRoot` is not 0. Currently, this is not enforced.
  - [ ] Test coverage
- Assure that `executionStateRoot` is not 0. Currently, this is not enforced.
  - [ ] Test coverage
- Ensure that the calculated `slotValue` matches the `messageRoot`.
  - [x] Test coverage

Negative behavior

- Should not be called multiple times for the same message.
  - [ ] Negative test
- No parameters should be reused for different messages.
  - [ ] Negative test

Function call analysis

- `recipient.call(receiveCall);`
  - **What is controllable?** `receiveCall` is partly controlled by the user.
  - **If return value controllable, how is it used and how can it go wrong?** N/A. The constraint here is that the `recipient` must implement the `ITelepathyHandler` interface. That is enforced through constructing the `receiveCall` variable with the selector of the `ITelepathyHandler.handleTelepathy` func-

tion.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Arbitrary code execution on the target `recipient` contract. Should be safe, as the `recipient` contract must implement the `ITelepathyHandler` interface.

5.4 File: TelepathyAccess.sol

Function: `freezeAll()`

Freezes messages from all chains.

Branches and code coverage (including function calls)

Intended branches

- Change the frozen state of all chain IDs to true.
  ☑ Test coverage
- Assumes all had not been frozen beforehand (by default they are not frozen).
  ☑ Test coverage

Negative behavior

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

Function: `freeze(uint32 chainId)`

Freezes messages from the specified chain.

Inputs

- `chainId`
  - **Control**: Full control.
  - **Constraints**: Assumes it is a valid chain ID; no other constraints.
  - **Impact**: The `chainID` will be frozen.

Branches and code coverage (including function calls)

Intended branches

- Change the frozen state of the chain ID to true.
  ☑ Test coverage
- Assumes it was not frozen beforehand (by default it is not frozen).

☑ Test coverage

Negative behavior

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

Function: `setLightClientAndBroadcaster(uint32 chainId, address lightclient, address broadcaster)`

Sets the light client contract and broadcaster for a given `chainId`.

Inputs

- `chainId`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: The mappings `lightClients` and `broadcasters` are updated with the new values at the given `chainId`.
- `lightclient`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: The mappings `lightClients` will be updated with the new `lightclient` value at the given `chainId`.
- `broadcaster`
  - **Control**: Full control.
  - **Constraints**: None.
  - **Impact**: The mappings `broadcasters` will be updated with the new `broadcaster` value at the given `chainId`.

Branches and code coverage (including function calls)

Intended branches

- Should check that the `chainId` exists in the `sourceChainIds` array. This could be done more easily through checking whether the `lightClients` or `broadcasters` mappings have a value at the given `chainId` instead of the current for loop.
  ☐ Test coverage
- Should check that the `lightclient` and `broadcaster` addresses are not the zero address.
  ☐ Test coverage
- Assumes that updating the mappings will not cause any issues down the line and that the function should not only be called once for each `chainId`.

☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the timelock.
  ☑ Negative test
- Should not re-add a `chainId` that already exists in the `sourceChainIds` array.
  ☐ Negative test

Function: `setSendingEnabled(bool enabled)`

Allows the owner to control whether sending is enabled or not.

Inputs

- `enabled`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: Whether sending is enabled or not.

Branches and code coverage (including function calls)

Intended branches

- Should change the value of `sendingEnabled` to `enabled`.
  ☑ Test coverage

Negative behavior

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

Function: `unfreezeAll()`

Unfreezes messages from all chains.

Branches and code coverage (including function calls)

Intended branches

- Should set the frozen flag to false for all chains.
  ☑ Test coverage

Negative behavior

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

☑ Negative test

Function: `unfreeze(uint32 chainId)`

Unfreezes messages from the specified chain.

Inputs

- `chainId`
  - **Control**: Full control.
  - **Constraints**: Assumes it is a valid chain ID; no other constraints.
  - **Impact**: The `chainID` will be unfrozen.

Branches and code coverage (including function calls)

Intended branches

- Change the frozen state of the chain ID to false.
  ☑ Test coverage
- Assumes it was frozen beforehand (by default it is not frozen).
  ☑ Test coverage

Negative behavior

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

5.5 File: TelepathyHandlerUpgradeable.sol

Function: `handleTelepathy(uint32 _sourceChainId, address _senderAddress, byte[] _data)`

When the TelepathyRouter executes a message, this handler will be called on the recipient address along with the user-provided data. The bytes4 selector is then returned for the caller to ensure the correct handler was called.

Preconditions

The `msg.sender` must be the `_telepathyReceiever`.

Inputs

- `_sourceChainId`

  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: Will be passed through to the implementation.
- `_senderAddress`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: Will be passed through to the implementation.
- `_data`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: Will be passed through to the implementation.

Branches and code coverage (including function calls)

Intended branches

- The implementation method is called and the selector returned.
  - ☐ Test coverage

Negative behavior

- Caller is not the `_telepathyReceiever`.
  - ☐ Negative test

Function: `__TelepathyHandler_init(address telepathyReceiever)`

Sets up the `telepathyReceiever` for an upgradable contract.

Preconditions

Can only be called by an `initializer` function.

Inputs

- `telepathyReceiever`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: The address will be able to call the `handleTelepathy` function.

Branches and code coverage (including function calls)

Intended branches

- The `telepathyReceiever` is set correctly.
  ☐ Test coverage

Negative behavior

- The calling function is not an initializer.
  ☐ Negative test

5.6 File: TelepathyHandler.sol

Function: `handleTelepathy(uint32 _sourceChainId, address _senderAddress, byte[] _data)`

When the TelepathyRouter executes a message, this handler will be called on the recipient address along with the user-provided data. The bytes4 selector is then returned for the caller to ensure the correct handler was called.

Preconditions

The `msg.sender` must be the `_telepathyReceiever`.

Inputs

- `_sourceChainId`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: Will be passed through to the implementation.
- `_senderAddress`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: Will be passed through to the implementation.
- `_data`
  - **Control**: Full control.
  - **Constraints**: No checks.
  - **Impact**: Will be passed through to the implementation.

Branches and code coverage (including function calls)

Intended branches

- The implementation method is called and the selector returned.
  ☑ Test coverage

Negative behavior

- Caller is not the `_telepathyReceiever`.
  - ☐ Negative test

5.7 File: TelepathyRouter.sol

Function: `initialize(uint32 _sourceChainIds, address _lightClients, address _broadcasters, address _timelock, address _guardian, bool _sendingEnabled)`

Initializes the contract and the parent contracts once.

Inputs

- `_sourceChainIds`
  - **Control**: Full.
  - **Constraints**: N/A; should check that the array is the same length as the other arrays.
  - **Impact**: The source chain IDs.
- `_lightClients`
  - **Control**: Full.
  - **Constraints**: N/A; should check that the array is the same length as the other arrays.
  - **Impact**: The addresses of the light clients
- `_broadcasters`
  - **Control**: Full.
  - **Constraints**: N/A; should check that the array is the same length as the other arrays.
  - **Impact**: The addresses of the broadcasters.
- `_timelock`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: The address that will receive admin and timelock roles.
- `_guardian`
  - **Control**: Full.
  - **Constraints**: N/A.
  - **Impact**: The address that will receive the guardian role.
- `_sendingEnabled`
  - **Control**: Full.

- **Constraints**: N/A.
- **Impact**: Whether or not sending is enabled.

Branches and code coverage (including function calls)

Intended branches

- Should initialize the contract and call the underlying initializer functions.
  - ☑ Test coverage
- Call `__UUPSUpgradeable_init()`.
  - ☑ Test coverage
- Call `_AccessControl_init()`.
  - ☑ Test coverage
- Call `_ReentrancyGuard_init()`.
  - ☑ Test coverage
- Ensure that the arrays are the same length.
  - ☑ Test coverage

Negative behavior

- Should not be callable twice.
  - ☑ Negative test
- Should not leave any variables uninitialized.
  - ☐ Negative test

6 Audit Results

At the time of our audit, the code was not deployed to mainnet EVM.

During our audit, we discovered four findings. Of these, one was high risk, two were medium risk, and the remaining finding was of low risk. Succinct acknowledged all findings and implemented fixes.

6.1 Disclaimers

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 additional code added to the assessed project after the audit version of 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 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.

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.