macro

Connext A-1

Security Audit

January 31, 2023
Version 1.0.0

Presented by 0xMacro

Table of Contents

Introduction

This document includes the results of the security audit for Connext's smart contract code as found in the section titled ‘Source Code’. The security audit was performed by the Macro security team from November 21, 2022 to December 30, 2022.

The purpose of this audit is to review the source code of certain Connext Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro’s review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	1	-	-	1
High	4	-	-	4
Medium	4	1	-	3
Low	5	4	-	1
Code Quality	9	2	-	7
Gas Optimization	5	3	-	2

Connext was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Telegram with the Connext team.
- Connext documentation.

Source Code

The following source code was reviewed during the audit:

- **Repository:** nxtp
- **Commit Hash:** `5bb508f6ac4ad1572cca74ef297c2f57260e700c`

Specifically, we audited the messaging layer contracts within this repository:

./packages/deployments/contracts/contracts/messaging/*

Contract	SHA256
messaging/MerkleTreeManager.sol	`b04e5f16da3c2e690dea5e7478895e9ebbf20688ca27356c75d2dad2b5fa9c83`
messaging/RootManager.sol	`ac7a3c5f9d087585984ceeeb68073897c3c32150c9bbc456bd09fc7829beb49e`
messaging/WatcherClient.sol	`a6b25f07ae326a0c990cc53aa82e112e88260025e2989627cb863870f446131c`
messaging/WatcherManager.sol	`3fcd1ffad12481e534c05b62ece31ee02109e16a144a6db94d72c6428261bdc5`
messaging/connectors/ConnectorManager.sol	`78a29659b3f55e4ac2f5b6a0f489048ea552c77b8371530a21bc255c6c5515f5`
messaging/connectors/Connector.sol	`8293745321f85eae032d63b8d3170c64509e3443844aee7aa0ad54b6c462fb34`
messaging/connectors/GasCap.sol	`d73ff15a5409670d4b1d0272168fb17a13659810057d201ce1076b5c57001f42`
messaging/connectors/HubConnector.sol	`0d138e14b7b8c80b54463580155c834af445580a838c70cb86a13a20ff1ef226`

messaging/connectors/SpokeConnector.sol	`fd7371b75d87409264673fb242cd1c7f5554ac975241451e71ff2eba43bd58bb`
messaging/interfaces/IConnectorManager.sol	`34583a69beb6430e1d3a563e2672bbb64e7316d8eb25ca2bbd62868b8cd77303`
messaging/interfaces/IConnector.sol	`18e69eed4b7226e8650fa078fab7cc47600f4bdd24652ffb2e38d23c3e7fc5a6`
messaging/interfaces/IHubConnector.sol	`2bf30b5788b487ddaa1e44333f999aa16ba71e7ab5ef4c476e3905bd3f00a53a`
messaging/interfaces/IMessageRecipient.sol	`cd9c247436a3baeca59201b1cab92367da901b383b7fbeacb48fb6f9acb8b421`
messaging/interfaces/IOutbox.sol	`18ee786f0d242a7bcabb9ff77d39c959a2c65be034dcfab007f56b54f32c3f8a`
messaging/interfaces/IRootManager.sol	`d1dcb24dae1549066036e00c9eac5293411dd899177295f6f401031e624af3be`
messaging/libraries/DomainIndexer.sol	`288a2eae98b98f448a1784d9e6364be1a4af2c143e5356aa0406dad85af3e525`
messaging/libraries/MerkleLib.sol	`cf54d747e95bf480174358083973c2d6debe35258373ca347713acbccd352671`
messaging/libraries/Message.sol	`94a5fac9b6cb07b4a1cd073d3dddd87daa3a685e70f3cd3b75904447b3901318`
messaging/libraries/Queue.sol	`ff36dbfcbbd5d0e9424c5bd1444019bb2cc0ca44433e6c4087a280340cb0cf43`
messaging/libraries/RateLimited.sol	`6662c61c68e2486ed87256592563fb276aac74afc9415323115243cce850845c`
messaging/connectors/arbitrum/ArbitrumHubConnector.sol	`87d280e2d06519059325ab834b9900419746856d85b60787e8a9c93fcb4e4fc6`

messaging/connectors/arbitrum/ArbitrumSpokeConnector.sol	`14b7f477967ed36f59a4064105fa205a5bd285c36f404d9356f118f9b04c2c3d`
messaging/connectors/gnosis/GnosisBase.sol	`ed253029ffb2eca8f4d8f1a226fa0610f32bb6f418c2acfd53ad64ed81cdb8a4`
messaging/connectors/gnosis/GnosisHubConnector.sol	`54cf16bff7396f71e8e7d7548116909d42eb9a7b49295e45c2616a4b1c8cce28`
messaging/connectors/gnosis/GnosisSpokeConnector.sol	`fe3cedb8caddd00e09fd5a9d7c4c53725e1866f9921fe8cc64e54d6190701b31`
messaging/connectors/mainnet/MainnetSpokeConnector.sol	`6a3b62900ecdb67676093b6af8c8e3361caf30448b7fe40b53419e6bcfb58f17`
messaging/connectors/multichain/BaseMultichain.sol	`d8b354631ffbf6f36df45295a363994c18a32b160ab3fc7a973b6a1993cd1e14`
messaging/connectors/multichain/MultichainHubConnector.sol	`7d3456e8f38b28721aed2d6c17525706de7263be640d5c44a4d380084ab942c6`
messaging/connectors/multichain/MultichainSpokeConnector.sol	`51da17d1fbc15abd0945a40a0c872dcc19b2f3ce5edf82a8f63290d1cb6823da`
messaging/connectors/optimism/BaseOptimism.sol	`8ba3dc99a2bb0fa668fdec812a3c35115c21ae78207c1e0056b7b72e9507960e`
messaging/connectors/optimism/OptimismHubConnector.sol	`55c1f9ab5be09dba9418f713e50ad97e51f5f0aec24f412bdd2acb3c153e82fd`
messaging/connectors/optimism/OptimismSpokeConnector.sol	`08dae0f5c8060dcf44c840f01eff9373e81c5cd4501204fa7bd379dc089318e4`
messaging/connectors/polygon/PolygonHubConnector.sol	`dfab6c988dfed12aa0b8c13b11fae4d04a665c4642405f50d3c60deaa78a9aad`
messaging/connectors/polygon/PolygonS	`e22fba656c80a81bd879fd73ca0026db`

pokeConnector.sol	`61e13f0b094d223ad782b36b15d6377e`
messaging/connectors/zksync/ZkSyncHubConnector.sol	`a95e5d3d8f09dd15698bca8e37764d809caab057fcc863993c568b7e6f1b7a77`
messaging/connectors/zksync/ZkSyncSpokeConnector.sol	`642ddae5ca662fe45a10d26c46c4aff41ac0038372756e67223876c836c6622a`
messaging/interfaces/ambs/GnosisAmb.sol	`92b521a95f598ce0f877a5b5940b0a637c24c3d4274ff3b61111b72c4d5bb8cd`
messaging/interfaces/ambs/Multichain.sol	`e258ec102a07a48e83a5fdfddbee8fb844fcc5ce7323597c45f22db299566bcb`
messaging/connectors/optimism/lib/BytesUtils.sol	`11d50d23ed041d8a6c49b34619b1e599c20bfc3aba3456d526e0c4df8426f0b6`
messaging/connectors/optimism/lib/MerkleTrie.sol	`98a171631172141e2b4e02ff8e015301aa1e5f0e58845d6a1d327cb290f73724`
messaging/connectors/optimism/lib/OVMCodec.sol	`8223ba09aba13533219ed1c9cb6d396ef0590b0d0ee32845466aab7bf0e67607`
messaging/connectors/optimism/lib/PredeployAddresses.sol	`7e6adeadf5623b8f89e559174c9f9a55b9b412c864b152fa48d15882df4a4ae0`
messaging/connectors/optimism/lib/RLPReader.sol	`76e76bb2aa3f12ce7d52fd69f6733288ce5473819ca36e6a3300c781fa090224`
messaging/connectors/optimism/lib/SecureMerkleTrie.sol	`2d473d117746578acfe2c88205f65241be2ce0f18a8794fbd441080cfe3f615b`
messaging/connectors/polygon/lib/ExitPayloadReader.sol	`ae7e13370677a5df6786dd121809840f8e22982fa5f9bb8ebde204c63c21f618`
messaging/connectors/polygon/lib/MerklePatriciaProof.sol	`b70d7ffd7d5ab3007ce6e13c57a7cd304180ab3c3864ec6cbfdb0ba3fc2d1f59`

messaging/connectors/polygon/lib/Merkle.sol	`f187ccc12759743534e57730ae104b3aa9b7aaf59a0df58b7c7bdcf7fd5f2675`
messaging/connectors/polygon/lib/RLPReader.sol	`e7edbbe6035d5c8ffb8b8ec1b2003af590efef17c35de7004239476d96351a94`
messaging/connectors/polygon/tunnel/FxBaseChildTunnel.sol	`d63f27b678da5e58ab1f6335570a2f4c5620002496a576dd5549b51f4e941305`
messaging/connectors/polygon/tunnel/FxBaseRootTunnel.sol	`1a2f3af04b7ec69ed3c3b95c76663c69de4d2c15aa6dc511560241a908b89699`
messaging/interfaces/ambs/arbitrum/ArbitrumL2Amb.sol	`12b515cf01596623a6b28a8c780674f857ac9fadf70b1e044a59e6d022e2a2b2`
messaging/interfaces/ambs/arbitrum/IArbitrumInbox.sol	`405f2b482c0bbc02345474728c02f80995728744d75287ee10be8eabed273c80`
messaging/interfaces/ambs/arbitrum/IArbitrumOutbox.sol	`643d51f5ea21073a79a035b41e20d009033f726a59fb97e507a93bc7cd4faaef`
messaging/interfaces/ambs/arbitrum/IArbitrumRollup.sol	`8652b5338730072da4627db8a4860d0b879660a8002bfa9130bfc4ac3466e804`
messaging/interfaces/ambs/optimism/IStateCommitmentChain.sol	`6f6624ca88102449213de14caf004af12784419802bee31c80548f9f7eb23599`
messaging/interfaces/ambs/optimism/OptimismAmb.sol	`1ef13616e80e6337a0dc0c72be02e9ae47f5b39b451ccfc36b4b2f04a1d3cdf1`

Note: This document contains an audit solely of the Solidity contracts listed above. Specifically, the audit pertains only to the contracts themselves, and does not pertain to any other programs or scripts, including deployment scripts.

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

~~C-1~~	Lack of access control on `RelayerProxyHub.propagate()` would allow anyone to drain the `RelayerProxyHub` contract.
~~H-1~~	RootManager.propagate susceptible to DoS
~~H-2~~	Watcher may halt root propagation due to incorrect `removeDomain` implementation
~~H-3~~	Messages with insufficient gas can halt processing on Gnosis Bridge
~~H-4~~	`SpokeConnector` swap allows replay of messages
~~M-1~~	Unspent assets can get stuck in the RootManager contract
~~M-2~~	`SpokeConnector` swap can result in duplicate messages
~~M-3~~	Missing functionality in ZkSyncSpokeConnector
M-4	SpokeConnector’s `proveAndProcess()` is susceptible to griefing
L-1	Incorrectly sent assets to GnosisHubConnector will be stuck
~~L-2~~	Assets sent to ZkSyncHubConnector and MultichainHubConnector may get stuck
L-3	Missing event emission in MainnetSpokeConnector and Multichain Connectors
L-4	Hardcoded gas values in SpokeConnector are potentially problematic
L-5	`PROCESS_GAS` guard insufficient for a batch of messages
~~Q-1~~	Make gasCap variable public
Q-2	Confusing usage of gasCap mechanism
Q-3	ZkSync AMB integration will require changes
~~Q-4~~	SenderAdded and SenderRemoved events emitted when not needed

~~Q-5~~ Update _processMessage in MainnetSpokeConnector to revert since it is not used

~~Q-6~~ Override processMessage in Multichain Connectors to revert since it is not used

~~Q-7~~ Document removePendingAggregateRoot use

~~Q-8~~ Documentation improvements

~~Q-9~~ `Check Effect Interaction` not followed for `send()` of spoke connector

G-1 Duplicate sender verification in ZkSyncSpokeConnector

~~G-2~~ Duplicate require condition in MainnetSpokeConnector

G-3 Replace sentMessageRoots mapping with the uint256 variable

~~G-4~~ Redundant increments for `removedCount` in `dequeueVerified`

G-5 Redundant copying of the arrays in `dequeueVerified`

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:
    - How bad things can get (for a vulnerability)
    - The significance of an improvement (for a code quality issue)
    - The amount of gas saved (for a gas optimization)
2. The high/medium/low **likelihood** of the issue:
    - How likely is the issue to occur (for a vulnerability)
3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client **must** fix the issue, no matter what, because not fixing would mean **significant funds/assets WILL be lost.**
(H-x) High	We recommend the client **must** address the issue, no matter what, because not fixing would be very bad, *or* some funds/assets will be lost, *or* the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to **seriously consider** fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albiet not in an existential manner.
(L-x) Low	The risk is small, unlikely, or may not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

~~C-1~~ Lack of access control on `RelayerProxyHub.propagate()` would allow anyone to drain the `RelayerProxyHub` contract.

TOPIC	STATUS	IMPACT	LIKELIHOOD
Access Control	Fixed	High	High

In `RelayerProxyHub` contract, `propagate()` is an external function without any access control. Hence, anyone can call it with an arbitrary relayer fee and valid root as input and withdraw all assets from the contract via `transferRelayerFee()`.

Remediations to Consider:

Consider restricting access to the relayers only.

~~H-1~~ RootManager.propagate susceptible to DoS

TOPIC	STATUS	IMPACT	LIKELIHOOD
Denial of Service	Addressed	High	Medium

In the RootManager contract, `propagate()` function is responsible for distributing aggregate roots to all connectors for corresponding domains. Implementation of this function does not revert if during interaction with a specific connector error is raised. This prevents a single inaccessible chain from causing a complete bridge halt. Particular functionality was introduced to fix a previously reported issue in Spearbit audit.

When `propagate()` is called a set of initial validation checks is performed. If these checks are successful `lastPropagatedRoot` value will be set. In the rest of `propagate()` function even if interaction with all connectors fails `propagate()` function will not revert and the `lastPropagatedRoot` value will remain set. This means that follow-up invocations of `propagate()` function will not be possible until `_aggregateRoot` is updated. This is indeed the original intention of this code.

```
validateConnectors(_connectors);

uint256 _numDomains = _connectors.length;
// Sanity check: fees and encodedData lengths matches connectors length.
require(_fees.length == _numDomains && _encodedData.length == _numDomains,

// Dequeue verified roots from the queue and insert into the tree.
(bytes32 _aggregateRoot, uint256 _count) = dequeue();

// Sanity check: make sure we are not propagating a redundant aggregate ro
require(_aggregateRoot != lastPropagatedRoot, "redundant root");
lastPropagatedRoot = _aggregateRoot;
```

However, since `propagate()` function can be invoked by anyone, as it doesn't have any access control limits, a malicious actor can invoke this function with inappropriate arguments. For example, by providing encoding data of unexpected length. This will cause most if not all connectors to revert when invoked. For example, processing will fail for:

- PolygonHubConnector in `_sendMessage()` function when `encodedData.length` is not 0
- ZKSyncHubConnector when `encodedData.length` is not 32
- BaseMultichain in `_sendMessage()` when `encodedData.length` is not 0

- GnosisBase in `_getGasFromEncoded()` when `encodedData.length` is not 32
- ArbitrumHubConnector in `_sendMessage()` when `encodedData.length` is not 96

As a result of malicious actor actions, aggregate roots won't be propagated to destination domains. The system may recover if a new root is added and `propagate()` is called with proper inputs. However, an attacker can front-run these calls with their own invocation with adverse inputs.

Remediations to Consider:

1. Remove the sanity check present at L201 of RootManager. This would keep `propagate()` permission-less, but there could be duplicate RootPropagated events.

   ```
   require(_aggregateRoot != lastPropagatedRoot, "redundant root");
   ```

2. Restrict access to `propagate()` function to relayers only.

RESPONSE BY CONNEXT

> We implemented an alternative fix to track the lastPropagatedRoot by domain, that way you could call the same function multiple times if a single domain failed. The spoke connectors will also revert if a duplicate aggregate root is received, so this seems a cleaner fix

---

~~H-2~~ Watcher may halt root propagation due to incorrect `removeDomain` implementation

TOPIC	STATUS	IMPACT	LIKELIHOOD
Protocol Design	Fixed	High	High

`RootManager` inherits from `DomainIndexer`, which allows watchers to remove connectors through `RootManager.removeConnector` > `DomainIndexer.removeDomain`. There are three data structures associated with this method: `domains[]`, `connectors[]`, and `domainToIndexPlusOne`. `domainToIndexPlusOne` maps the domain to its index in the array. These three data structures need to be updated correctly and always in sync for domain management functionality to work properly.

However, the current `removeDomain()` function implementation is incorrect when the domain to be removed is not the last. The `removeDomain` function swaps the last element from the `domains[]` and `connectors[]` array with the to-be-removed element and pops the last element. Also, the `domainToIndexPlusOne` mapping is cleared for the to-be-removed element. But the update to `domainToIndexPlusOne` for the swapped last connector is missing. As a result, the swapped domain will have an incorrect index which will cause incorrect system behavior.

Due to the incorrect index for the swapped domain, it will not be possible for the corresponding connector to invoke `RootManager.aggregate()` function. When invoked, `aggregate()` function will fail in `onlyConnector(_domain)` modifier as `getConnectorForDomain()` will revert due to an unexpected connector address or due to an index array out-of-bounds error. Therefore message propagation originating from this particular domain will be halted.

To recover from the data corruption contract upgrade is necessary. Current contract functionality does not provide means to reset data structures to a valid state. For example, a subsequent call to `removeConnector()` for the now corrupted domain may have an unexpected effect in removing a connector for a totally different domain due to an incorrect `domainToIndexPlusOne` record, further exacerbating data corruption. Also, a subsequent call to `addConnector()` will fail since the domain already exists.

In summary, *watchers* accidentally or intentionally may halt root propagation for one or more domains due to incorrect `removeDomain()` function implementation, and the current system does not provide means to the *owner* to recover from this issue without RootManager contract redeployment.

Remediations to Consider:

Consider updating `domainToIndexPlusOne` mapping to reflect the new position of swapped last element.

~~H-3~~ Messages with insufficient gas can halt processing on Gnosis Bridge

TOPIC	STATUS	IMPACT	LIKELIHOOD
Denial of Service	Fixed	High	Medium

To propagate a root from Mainnet to the Gnosis chain, *anyone* can invoke `propagate()` function on the RootManager with the appropriate value in `encodedData`. This variable encodes the maximum gas to be used to process this message at the destination (Gnosis chain). `propagate()` triggers `HubConnector.sendMessage()` which itself relies on `GnosisHubConnector._sendMessage()` to deliver a message to the Gnosis AMB by calling `requireToPassMessage()`.

```
GnosisAmb(AMB).requireToPassMessage(
  mirrorConnector,
  abi.encodeWithSelector(Connector.processMessage.selector, _data),
  _getGasFromEncoded(_encodedData)
);
```

On the other hand, to send a new origin root from the Gnosis chain to Mainnet, *anyone* can invoke `send()` function on the corresponding GnosisSpokeConnector with the appropriate value in `_encodedData`. As in the previous case, this variable is used to define the maximum gas to be used to process this message at the destination (Mainnet). `send()` function relies on `GnosisSpokeConnector._sendMessage()` to deliver a message to the GnosisAmb by calling `requireToPassMessage()`.

The only validation in Connext contracts mentioned above and related to encodedData is to verify that `encodedData.length` is 32 bytes long so that it can be decoded as uint256 which Gnosis AMB expects. Also, it validates that provided gas is less than the gasCap. However, within Gnosis AMB itself - more precisely `MessageDelivery._sendMessage()`, there is additional validation that this value should be bigger than the constant `MIN_GAS_PER_CALL` which is 100.

```
function requireToPassMessage(address _contract, bytes memory _data, uint2
  return _sendMessage(_contract, _data, _gas, SEND_TO_ORACLE_DRIVEN_LANE);
}

function _sendMessage(address _contract, bytes memory _data, uint256 _gas,
  internal
  returns (bytes32)
{
  // it is not allowed to pass messages while other messages are processec
  // if other is not explicitly configured
  require(messageId() == bytes32(0) || allowReentrantRequests());
  require(_gas >= MIN_GAS_PER_CALL && _gas <= maxGasPerTx());
...
```

If a message that is successfully posted from either side to the Gnosis AMB bridge has a gas set to value which is insufficient to guarantee proper processing, that message won’t be properly processed on the destination.

Due to the above, an attacker may invoke `RootManager.propagate()` on Mainnet or

`GnosisSpokeConnector.send()` on the Gnosis chain, and provide intentionally insufficient max gas in `encodedData`, e.g. value of 100. This amount of gas is not enough for roots to be processed at the destination, but it is enough to pass validation at Gnosis AMB. As a result, the attacker may halt processing on one or both sides of the chain. On the origin, the state will be updated correctly, but on the destination, there will be no updates.

Remediations to Consider

- Adding corresponding min gas limit on GnosisHubConnector and GnosisSpokeConnector that will guarantee successful processing on destination

---

H-4 `SpokeConnector` swap allows replay of messages

TOPIC	STATUS	IMPACT	LIKELIHOOD
Protocol Design	Fixed	High	Low

`Connext` enables replacing connectors without dropping the messages by maintaining the Merkle tree in a separate contract from actual connectors.

However, if it's done to any spoke connector in the current code's state, one can replay the already executed message on swapped spoke connector.

Spoke connectors record if the message is processed inside `messages` mapping, and doesn't allow to replay it through `calculateMessageRoot`.

```
function process(bytes memory _message) internal returns (bool _success) {
  ...
  messages[_messageHash] = MessageStatus.Processed;
  ...
```

```
}

function calculateMessageRoot(
  bytes32 _messageHash,
  bytes32[32] calldata _messagePath,
  uint256 _messageIndex
) internal view returns (bytes32) {
  // Ensure that the given message has not already been proven and process
  require(messages[_messageHash] == MessageStatus.None, "!MessageStatus.Nc
    ...
}
```

However, since this `messages` mapping resides in the spoke connector contract only, if it is swapped with a new one, the new one won't have the old state, allowing one to process an already executed message again.

Remediation to Consider

- Maintain `messages` mapping in a separate contract

RESPONSE BY CONNEXT

> acknowledged — currently when the liquidity layer is the only whitelisted sender, and there is protection against replays at that level, this attack should not be possible. however, this is a requirement to fix as more consumers use the messaging layer.

~~M-1~~ Unspent assets can get stuck in the RootManager contract

TOPIC	STATUS	IMPACT	LIKELIHOOD
Asset Recovery	Fixed	Medium	Medium

In the RootManager contract, callers of `propagate()` function submit assets necessary to cover fees for processing of `aggregateRoot` propagation on destination chains.

However, if interaction with a particular connector fails for any reason, `propagate()` function will still succeed, and sent assets will remain in the RootManager contract.

Currently, there is no mechanism in the RootManager contract to withdraw these unspent assets, and therefore they will remain stuck.

Remediations to Consider

Consider:

- refunding unspent assets to the caller
- introducing admin-controlled `withdrawFunds()` function

~~M-2~~ `SpokeConnector` swap can result in duplicate messages

TOPIC	STATUS	IMPACT	LIKELIHOOD
Protocol Design	Fixed	High	Low

`Connext` upgradeability approach relies on swapping contracts, such as SpokeConnector instances, in order to update code logic without dropping the messages. Most of the state is maintained in the Merkle tree in a separate contract from actual connectors, which facilitates upgrades using this approach.

However, one important part of the state is handled by the SpokeConnector contract itself. That is `nonces` mapping which ensures that messages originating from that particular domain and directed towards a concrete destination chain are unique. `nonces` variable is used in SpokeConnector in the `dispatch()` function as part of the message wrapper. A message with a wrapper is then formatted and hashed. The hash of the message is then used as an identifier within the system.

```
// Get the next nonce for the destination domain, then increment it.
uint32 _nonce = nonces[_destinationDomain]++;

// Format the message into packed bytes.
bytes memory _message = Message.formatMessage(
  DOMAIN,
  TypeCasts.addressToBytes32(msg.sender),
  _nonce,
  _destinationDomain,
  _recipientAddress,
  _messageBody
);

// Insert the hashed message into the Merkle tree.
bytes32 _messageHash = keccak256(_message);

// Returns the root calculated after insertion of message, needed for ever
// watchers
(bytes32 _root, uint256 _count) = MERKLE.insert(_messageHash);
```

When `SpokeConnector` is swapped, the state is reset which includes `nonces` mapping. As a result, it is possible that the same combination of parameters ends up in `Message.formatMessage()`. In this case, a new message with the same identifier will be inserted in the Merkle tree on the origin and propagated to the destination. However, on the destination, it won't be possible to process it as it would be considered a duplicate message. This would be most problematic in cases where the external protocol is using the `connext` as messaging bridge to do various defined actions on cross-chain destinations.

Remediation to Consider:

- Maintain `nonces` mapping outside of the SpokeConnector contract.

RESPONSE BY CONNEXT

> acknowledged — currently when the liquidity layer is the only whitelisted sender, and there is protection against duplicate hashes because of unique transfer ids (passed in message body) at the liquidity layer. unique ids generated here:

~~M-3~~ Missing functionality in ZkSyncSpokeConnector

TOPIC	STATUS	IMPACT	LIKELIHOOD
Protocol Design	Addressed	High	Low

ZkSync 2.0 peculiarity of preserving `msg.sender` for L1 to L2 calls introduces a need for additional functionality in ZkSyncSpokeConnector. When ZkSyncHubConnector initiates a call to the `processMessage()` function of ZkSyncSpokeConnector, the caller (`msg.sender`) on L2 will have the address of `mirrorConnector` .

Currently, the `processMessage()` function is protected by the `onlyAMB` modifier which performs a check that `msg.sender == AMB` . This check will be satisfied only when `AMB == mirrorConnector` .

In case, when the mirror connector contract address changes on L1, the contract owner of ZkSyncSpokeConnector, has the capability to update the `mirrorConnector` variable on L2 by calling `setMirrorConnector()` .

However, there is no corresponding functionality for the owner to update the value of the `AMB` variable. Therefore, in the case of the mirror connector contract address change on L1, it will not be possible to update and have a properly operating ZkSyncSpokeConnector.

Remediations to Consider:

- Add capability to update AMB variable on Connector
- Add capability to update AMB variable only on ZkSyncSpokeConnector
- Override `processMessage()` in ZkSyncSpokeConnector and change access control guard to compare `msg.sender` with `mirrorConnector` instead of `AMB`

RESPONSE BY CONNEXT

> Fixed this in a different way then suggested — made Connector.processMessage(...) (which holds the onlyAMB modifier) a virtual function, which is overridden in the ZKSyncSpokeConnector without the modifier. This means the _processMessage msg.sender check in the spoke connector is preserved, while the check that the AMB is the sender. Making Connector.processMessage virtual was done to fix arbitrum aliasing

---

M-4 SpokeConnector's `proveAndProcess()` is susceptible to griefing

TOPIC	STATUS	IMPACT	LIKELIHOOD
Denial of Service	Acknowledged	Medium	Medium

In the SpokeConnector contract, the `proveAndProcess()` function, which is responsible for executing messages associated with propagated roots, can be called by anyone. This function is expected to be called by relayer infrastructure in normal circumstances, but it also allows permissionless access in order to reduce centralization risk. This function also supports handling a batch of messages for execution as a performance optimization.

However, when the relayer submits a batch to `proveAndProcess()`, an attacker or interested 3rd party may front-run that transaction with a call to `proveAndProcess()` with the message that is the last message in the batch. That last message from the attacker's transaction will be successfully processed, but the batch transactions submitted by the relayer will fail at the end of message array processing.

Based on the provided information, there is no complex logic on retry logic on the relayer execution side for failed `proveAndProcess()` call. Therefore, an attacker may cause disruption in the processing of messages on the destination chain, unnecessary gas expenses related to the relayer infrastructure, and reduce or practically remove performance optimization benefits of processing a batch of messages in a single call.

Remediations to Consider:

- Verify that messages provided to `proveAndProcess()` are in an adequate state before processing any of them, or
- Limit access to `proveAndProcess()`

RESPONSE BY CONNEXT

> acknowledged, but likely will not fix. fixing this issue requires offchain logic (which will not be enforceable as this is open to any relayer network), or instituting a relayer whitelist at the messaging layer. the worst case is the optimization of batch processing falls back to the simple case of processing a single message

---

L-1 Incorrectly sent assets to GnosisHubConnector will be stuck

TOPIC	STATUS	IMPACT	LIKELIHOOD
Asset Recovery	Acknowledged	Medium	Low

GnosisHubConnector contract inherits functionality from HubConnector. This inherited functionality includes the `sendMessage()` function which is `payable`.

```
function sendMessage(bytes memory _data, bytes memory _encodedData) extern
  _sendMessage(_data, _encodedData);
  emit MessageSent(_data, _encodedData, msg.sender);
}
```

However, GnosisHubConnector doesn't have any use for assets sent to it, nor any means to withdraw these assets in case they are incorrectly sent to it from the RootManager contract.

Remediations to Consider:

- Override `sendMessage()` in GnosisHubConnector and remove `payable`, or
- Add means to rescue/withdraw assets from the GnosisHubConnector

RESPONSE BY CONNEXT

> acknowledged — this would require bad inputs to the fees argument in the propagate function. will fallback to the maxim of "dont send stupid things" instead of explicitly reverting here!

~~L-2~~ Assets sent to ZkSyncHubConnector and MultichainHubConnector may get stuck

TOPIC	STATUS	IMPACT	LIKELIHOOD
Asset Recovery	Addressed	Medium	Low

In the ZkSyncHubConnector contract, the `_sendMessage()` function forwards the `fee` amount of assets to ZkSync AMB. `fee` amount of assets is determined based on `msg.value` and value of configured `gasCap` on the ZkSyncHubConnector contract.

```
// In ZkSyncHubConnector
uint256 fee = _getGas(msg.value);

// In GasCap contract
function _getGas(uint256 _gas) internal view returns (uint256) {
  if (_gas > gasCap) {
    _gas = gasCap;
  }
  return _gas;
}
```

When `msg.value` is less or equal to configured `gasCap` all assets will be forwarded to the ZkSync AMB. However, when `msg.value` is greater, then only the `gasCap` of assets will be forwarded while `msg.value - gasCap` of assets will remain stored in ZkSyncHubConnector. Since currently there is no mechanism to withdraw this remaining amount of assets they will remain stuck in the contract.

The same issue is present MultichainHubConnector, more precisely in the BaseMultichain contract `_sendMessage()` function.

```
// Get the max fee supplied
uint256 supplied = _getGas(msg.value); // fee paid on origin chain, up to
```

Previous related issue.

Remediations to Consider:

- Add means to rescue/withdraw assets from the ZkSyncHubConnector, and
- Add means to rescue/withdraw assets from the MultichainHubConnector

RESPONSE BY CONNEXT

> ended up going with a simpler solution — allowing admin withdrawals on the connector directly. there is no incentive to use funds above the cap, and they are retrievable by the owner where implemented. this also mirrors existing spoke connector functionality

---

L-3 Missing event emission in MainnetSpokeConnector and Multichain Connectors

TOPIC	STATUS	IMPACT	LIKELIHOOD
Events	Acknowledged	Low	High

MainnetSpokeConnector

The MainnetSpokeConnector contract differs from other SpokeConnectors since it integrates both Hub and Spoke functionality within a single contract. As a result, some functions such as the `processMessage()` function, are not needed, and even though they are present, they are not called since there is no AMB.

However, this means that corresponding event emissions are not happening. For example, the `MessageProcessed` event in `Connector.processMessage` will never be emitted in the case of MainnectSpokeConnector. This inconsistent behavior with how other Connectors function may impact off-chain tools and systems relying on these events. Related issue.

Remediations to Consider:

- Add `MessageProcessed` event to the `_sendMessage()` function, and
- Move `MessageSent` event emission in `sendMessage()` top-level function to come before `_sendMessage()` call which will generate the corresponding `MessageProcessed` event.

MultichainHubConnector and MultichainSpokeConnector

Multichain connectors use `anyExecute` instead of `Connector.processMessage` for processing messages on both the hub and spoke side.

```
function anyExecute(bytes memory _data) external returns (bool success, by
  _processMessage(_data);
}
```

However since the `MessageProcessed` event is not emitted here, it creates an inconsistency with respect to other connectors.

Remediations to Consider:

- Add `MessageProcessed` event to the `anyExecute()` function

RESPONSE BY CONNEXT

> yeah i think we are okay with this! we use the more specific function, and processing message where they are functionally “processed” seems logically consistent

---

L-4 Hardcoded gas values in SpokeConnector are potentially problematic

TOPIC	STATUS	IMPACT	LIKELIHOOD
Protocol Design	Acknowledged	Medium	Low

Issue

SpokeConnector, while doing an external call on the destination, forwards only `PROCESS_GAS` for message execution on the destination, and keeps `RESERVE_GAS` to be able to complete the execution.

The issue with `PROCESS_GAS` and `RESERVE_GAS` is that they are immutable. Hence, if there is a change in gas cost for some opcode, the message processing that was working initially may fail. And since these two configuration values are immutable, it won't be possible to update these values.

Dependence on hardcoded gas costs isn't a good practice, they have changed in the past, and there is no guarantee they won't change in the future.

Informational

In the SpokeConnector constructor, provided `_processGas` and `_reserveGas` values need to be larger than 850_000 and 15_000 correspondingly. These hardcoded values may not be appropriate for chains with different opcode gas prices, such as ZkSync 2.0 (for which gas price tables are not yet available).

Remediations to Consider:

- Consider making both `PROCESS_GAS` and `RESERVE_GAS` mutable variables.

RESPONSE BY CONNEXT

> agree with acknowledge, the goal for connectors here is to keep operational

> costs as cheap as possible, meaning max immutables over mutables

---

L-5 `PROCESS_GAS` guard insufficient for a batch of messages

TOPIC	STATUS	IMPACT	LIKELIHOOD
Protocol Design	Acknowledged	Medium	Low

In the SpokeConnector contract, when the relayer submits a batch to `proveAndProcess()`, each message in the batch is processed individually in the underlying `process()` function. In this function, an external call is made with `PROCESS_GAS` forwarded to the callee. Before this call is performed there is a guard which checks the current gas left. This guard is effective when the `proveAndProcess()` executes a batch of size 1 and guarantees successful transaction completion independent of the status of the external call.

```
// A call running out of gas TYPICALLY errors the whole tx. We want to
// a) ensure the call has a sufficient amount of gas to make a
//    meaningful state change.
// b) ensure that if the subcall runs out of gas, that the tx as a whole
//    does not revert (i.e. we still mark the message processed)
// To do this, we require that we have enough gas to process
// and still return. We then delegate only the minimum processing gas.
require(gasleft() > PROCESS_GAS + RESERVE_GAS - 1, "!gas");
```

However, in the case when the batch size is bigger than 1, to guarantee the execution of `proveAndProcess()` for all messages without reverts, the caller needs to provide at least a `number of messages * (PROCESS_GAS + RESERVE_GAS - 1)`.

Otherwise, a first message in the batch can consume all of the `PROCESS_GAS` provided and cause the processing of messages that follow in the batch to fail due to not enough gas left to satisfy the guard condition listed above. In that case, a revert will be raised in the process function where this condition is checked and will cause the whole `proveAndProcess()` transaction to fail.

Remediations to Consider:

- Consider checking in the `proveAndProcess()` function that the amount of provided gas for processing a whole batch of messages is appropriate

RESPONSE BY CONNEXT

> The require is in the process function, which is called within a for loop in the top-level proveAndProcess. If there is insufficient gas for one of the messages, the whole call should revert. But yes, this does contradict the comment and safeguards in case the entire tx doesn't revert.

~~Q-1~~ Make gasCap variable public

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	Medium

In the GasCap contract, `gasCap` variable misses the visibility specifier. When visibility is not defined, it is `internal` by default, making it inaccessible to query it externally. Other connectors have similar variables with public visibility so to be consistent make gasCap public also.

Q-2 Confusing usage of gasCap mechanism

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Acknowledged	Medium

In the ZkSyncHubConnector contract, `_gasCap()` function and underlying `gasCap` variable represent submission cost capacity, while in the case of GnosisHubConnector, `gasCap` variable represents only one factor of submission cost, more precisely the maximum amount of gas to be used. Consider introducing a new variable/function to indicate different cap mechanics.

RESPONSE BY CONNEXT

> acknowledge — the consistent naming may not be suitable for all the ways its used but the mechanism can be shared (as it all provides some degree of protection from overpaying). would rather keep the logic in a single contract / place

---

Q-3 ZkSync AMB integration will require changes

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Acknowledged	Medium

The Mailbox contract in Zk Sync AMB is incomplete, and additional changes are to be expected. For example, the implementation of the l2TransactionBaseCost() function is marked as TODO, while comments in requestL2Transaction() seem to indicate that additional functionality will be added in the near future. Thus, when this happens, code on the Connext side which interacts with the ZkSync AMB will need to be reviewed and potentially updated.

~~Q-4~~ SenderAdded and SenderRemoved events emitted when not needed

TOPIC	STATUS	QUALITY IMPACT
Events	Fixed	Medium

In the SpokeConnector contract, in `addSender()` and `removeSender()` functions, there is no check if the operation to be performed will result in state changes. Therefore, events like SenderAdded and SenderRemoved will be emitted even if no state change happened, which may confuse off-chain monitoring tools and services.

~~Q-5~~ Update _processMessage in MainnetSpokeConnector to revert since it is not used

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	High

In MainnetSpokeConnector, `processMessage()` on Connector is not used but the underlying implementation in MainnetSpokeConnector `_processMessage()` does not revert.

~~Q-6~~ Override processMessage in Multichain Connectors to revert since it is not used

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	High

In MultichainHubConnector and MultichainSpokeConnector, `processMessage()` is not used. However, the underlying implementation `_processMessage()` is used by the `anyExecute()` function in the BaseMultichain contract. Consider overriding `processMessage()` in MultichainHubConnector and MultichainSpokeConnector with the implementation that reverts if invoked.

Q-7 Document removePendingAggregateRoot use

TOPIC	STATUS	QUALITY IMPACT
Documentation	Fixed	Medium

In SpokeConnector, `removePendingAggregateRoot()` represents a safety mechanism that can be used to protect from issues in the transport layer. However, it cannot be used as a fallback safety mechanism in the case when fraud detection has already failed on the mainnet. Therefore, document the capabilities and limitations of removePendingAggregateRoot so it is not possible to misunderstand the safety properties which it provides.

Q-8 Documentation improvements

TOPIC	STATUS	QUALITY IMPACT
Documentation	Fixed	Medium

- SpokeConnector
  - The top “@notice” comment refers to extending HubConnector which is incorrect as it inherits from Connector
  - Missing natspec for events. Only FundsWithdrawn event declaration has proper natspec
  - Missing indexed attributes for corresponding events (e.g. on Dispatch and Process for leaf)
  - Struct Proof declaration has incorrect natspec comment for the 2nd field - natspec refers to it as **proof** while the field name is a **path**
  - `allowlistedSenders` public variable doesn’t have a corresponding “@notice” natspec comment
  - inconsistent natspec comments for functions. Some are missing param definitions (e.g. `send()`) while others are missing return values descriptions (e.g. `dispatch()`)
  - incorrect natspec comment for `send()` function which is a copy of a natspec comment for a different function `outboundRoot()`
  - The “@notice” comment for `receiveAggregateRoot()` is confusing since we are not able to distinguish between these two invocation paths mentioned in the comment

~~Q-9~~ `Check Effect Interaction` not followed for `send()` of spoke connector

TOPIC	STATUS	QUALITY IMPACT
Code Quality	Fixed	High

```
function send(bytes memory _encodedData) external payable whenNotPaused ra

        // check
        require(sentMessageRoots[root] == false, "root already sent");
        ......
        // interaction
    **_sendMessage(_data, _encodedData);
                                        external call to AMB**

        // effect
    **sentMessageRoots[root] = true;
        .........**
  }
```

Since check effect interaction is not followed, one can reenter and send messages again; however, we couldn't find an incentive for one to do so and a AMB which would allow one to reenter, marking this as code quality.

Consider moving effect before an interaction to not have any exposure moving forward.

G-1 Duplicate sender verification in ZkSyncSpokeConnector

TOPIC	STATUS	GAS SAVINGS
Gas Optimisation	Acknowledged	Medium

ZkSync 2.0 preserves `msg.sender` value for L1 to L2 calls. This applies to the ZkSyncHubConnector’s call to `processMessage()` on ZkSyncSpokeConnector. As a result, the `mirrorConnector` and `AMB` variables must be set to the same value in ZkSyncSpokeConnector. Also due to this peculiarity, the `onlyAMB` modifier defined on `processMessage()` function and guard `_verifySender(mirrorConnector)` in `_processMessage()` function perform the same check. Consider removing `_verifySender(mirrorConnector)` in `ZkSyncSpokeConnector._processMessage()` as it is not needed.

~~G-2~~ Duplicate require condition in MainnetSpokeConnector

TOPIC	STATUS	GAS SAVINGS
Gas Optimisation	Fixed	Medium

In the MainnetSpokeConnector contract, in the `sendMessage()` function there is a check that `_encodedData.length` is 0. This same check is present in the internal `_sendMessage()` function which is called immediately after and does most of the processing. Consider removing the check in the top-level `sendMessage()` function.

G-3 Replace sentMessageRoots mapping with the uint256 variable

TOPIC	STATUS	GAS SAVINGS
Gas Optimisation	Acknowledged	Medium

Merkle tree count represents a continuously increasing counter. A new uint256 storage variable on SpokeConnector that relies on Merkle tree count can remove the need for `sentMessageRoots` mapping that is used in SpokeConnector `send()` function. Consider replacing it to reduce contract storage needs and corresponding gas usage.

```
// instead of following
bytes32 root = MERKLE.root();
require(sentMessageRoots[root] == false, "root already sent");
bytes memory _data = abi.encodePacked(root);
_sendMessage(_data, _encodedData);
sentMessageRoots[root] = true;
emit MessageSent(_data, _encodedData, msg.sender);

// consider following
(bytes32 root, uint256 count) = MERKLE.rootAndCount();
require(count > lastSent, "root already sent");
bytes memory _data = abi.encodePacked(root);
_sendMessage(_data, _encodedData);
lastSent = count;
emit MessageSent(_data, _encodedData, msg.sender);
```

RESPONSE BY CONNEXT

> acknowledged — generally, this will not be reliable once we move the tracking of roots sent to within the merkle tree manager

---

~~G-4~~ Redundant increments for `removedCount` in `dequeueVerified`

TOPIC	STATUS	GAS SAVINGS
Gas Optimisation	Fixed	Medium

`removedCount` is only used to identify if there are any removed items in the queue;

hence instead of incrementing it for each removed item, consider using a boolean identifier.

```
function dequeueVerified()
-------
while (!(first > last)) {
      bytes32 item = queue.data[first];
      if (!queue.removed[item]) {
        -------
      } else {
        // The item was removed. We do NOT increment the index (we will re
        unchecked {
          ++removedCount; // @audit you just need identifier
        }
      }
                ------
    }

    // Update the value for `first` in our queue object since we've dequeu
    queue.first = first;

    if (removedCount == 0) {
      return items;
    } else {
      -------
    }
```

G-5 Redundant copying of the arrays in `dequeueVerified`

TOPIC	STATUS	GAS SAVINGS
Gas Optimisation	Acknowledged	Medium

Inside `dequeueVerified`, `items` is defined to be of the length of the queue. If some positions of items remained unfilled due to removed items, the `items` array is currently copied into a new array called `amendedItems` upto `index`.

```
if (removedCount == 0) {
      return items;
    } else {
      // If some items were removed, there will be a number of trailing 0
      // from the array. Create a new array with all of the items up until
      bytes32[] memory amendedItems = new bytes32[](index); // The last `i
      for (uint256 i; i < index; ) {
        amendedItems[i] = items[i];
        unchecked {
          ++i;
        }
      }
      return amendedItems;
}
```

This is an anti-pattern; even if the removed item is one, the whole array is copied.

Instead, consider changing the length of the items array through the assembly.

`mstore(items, length)`

Disclaimer