OpenZeppelin | security

AVA Teleporter Audit

Ava Labs.

November 16, 2023

Table of Contents

Summary

Type	DeFi
Timeline	From 2023-10-02 To 2023-10-31
Languages	Solidity

Total Issues	28 (13 resolved, 7 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	2 (0 resolved, 1 partially resolved)
Low Severity Issues	11 (3 resolved, 2 partially resolved)
Notes & Additional Information	15 (10 resolved, 4 partially resolved)

Scope

We audited the ava-labs/teleporter repository at commit 253b833 and the ava-labs/subnet-evm repository at commit c354ad6.

In scope were the following files:

```
contracts/src/Teleporter
├── ITeleporterMessenger.sol
├── ITeleporterReceiver.sol
├── ReceiptQueue.sol
├── ReentrancyGuards.sol
├── SafeERC20TransferFrom.sol
└── TeleporterMessenger.sol

subnet-evm/contracts/contracts/interfaces
└── IWarpMessenger.sol
```

The example cross-chain applications (CCAs) included in the repository under the `contracts/src/CrossChainApplications` directory were reviewed and used as a reference for how the `TeleporterMessenger` contract is intended to be used. These examples were not explicitly audited and this report does not include any issues found while reviewing these examples.

Tests, dependencies, and other parts of the protocol have been left out of the audit. Knowledge of how the main Avalanche network works has been used to explore possible attack vectors but was not considered part of the scope.

System Overview

The Teleporter messenger creates a user-friendly layer to send arbitrary messages between subnets using the Warp messaging system. The Warp Messenger operates at the virtual machine (VM) layer and offers the most flexibility for moving a payload between subnets. The Teleporter messenger, however, makes use of this flexibility and adds limitations to prevent incorrect use, such as replayability or Sybil attacks on other subnets. As such, it is meant to be used as an intermediate contract between CCAs and the Warp Messenger and is not meant to be used with non-contract recipients.

When a user wants to send a message from the origin subnet through the CCAs' endpoint, the message is wrapped, given a message ID, stored, and sent to the Warp Messenger precompile. Next, a relayer bundles them together in a certain arbitrary order, gets a signature for each message, and then aggregates a BLS signature for the unsigned messages. The relayer then submits a transaction on the destination chain where the messages and signature are submitted in the Access List of the transaction, encoded as a precompile. Each message on the destination chain will contain a destination address and subnet as specified when sending the message. Due to their design, the `TeleporterMessenger` contracts in all subnets will be deployed under the same address using Nick's deployment method. Meaning, the destination address for the Warp message will be the same address as the `TeleporterMessenger` contract at the origin subnet.

Relayers are incentivized to process messages through an ERC-20 token fee that is paid for by the message sender. The message sender is able to specify which ERC-20 token they wish to pay for their message with and how much of that token to spend. A relayer is only able to redeem this fee after they have executed the message on the destination chain and a receipt has been received on the origin chain.

Due to the flexibility of the fee mechanism, it is expected that relayers will only relay messages that will be profitable for them to execute and for ERC-20 tokens that they have vetted. The message sender must decide what ERC-20 token to use and an appropriate amount to incentivize relayers with. The fee amount for a message can be increased after initial submission to further incentivize relayers if the message is not being relayed. The relayer asks for validation and when enough validators' signatures (weighted in stake) reach the threshold, the message can be delivered to the recipient on the destination subnet.

While the message is being submitted, the relayer calls the hook on the `TeleporterMessenger` contract in the same transaction to receive the message and tries to execute it. If, for some reason external to the relayer, the message cannot be executed, it is stored to be executed again in the future. However, as the relayer's task is done, a receipt is created to then be attached to a message going in the opposite direction. Thus, when it is delivered, the relayer can collect the fee stored in the origin `TeleporterMessenger` contract for its work.

The protocol also includes mechanisms to speed up the receipt movement (in case no messages are going in the opposite direction), retry methods for failed executions, and add more fees to a sent message, among other things.

Security Model and Trust Assumptions

Subnets are special chains in the Avalanche project that allow for the modification of how the chain works at the VM level. This means that users must be aware of the fact that gas costs, opcodes, behaviors, and other characteristics might vary from one subnet to another. This not only impacts the operational workflow but could also cause issues if CCAs are not completely tackling these changes on each chain.

Moreover, the Warp protocol is meant to be as low on restrictions as possible, delegating all the responsibility to both the `TeleporterMessenger` contract (to an extent) and the CCA. This would be in order to assert that the message is doing what it should be doing and cannot be used for anything else.

Due to the flexibility given to the Teleporter protocol by design, there is no allow list for the tokens that users could use for paying the fees to the relayer. This means that relayers must be aware of not redeeming their rewards when those are paid in potentially malicious tokens or tokens that might include an unexpected transfer hook (e.g., ERC-777).

Privileged Roles

The `TeleporterMessenger` contract does not have privileged roles that could restrict or pause its operation. However, there are diverse actors who are part of the whole process of

sending a message, who can impact the functionality of sending cross-chain messages. These are:

- Relayers:
  - They can choose which messages to relay and have no obligation to relay messages. To the best of our knowledge, there is no slashing for relayers who choose to not relay a message. Further, they may be restricted in which messages they relay depending on external factors such as if the message interacts with OFAC-sanctioned accounts.
  - They can also sort the messages at will, meaning that messages will be added in a potentially different order from the one mined at the origin. Moreover, the reception at the destination can also be done in a different order from the one in the Warp message.
  - If a relayer can control a message's output at a certain destination contract (e.g., the criteria for the message are not met afterwards), it will have an incentive for failing such a call, reverting the reception of the message early, and using less gas while keeping the entirety of the fee assigned at the origin.
- Validators:
  - Validators are needed to sign messages coming from the relayers' bundles. However, if for some reason they are undergoing downtime, enough stake would not be achieved and messages might not meet the required minimum to be signed and delivered to the destination chain.
- Other users:
  - Can add fees and potentially malicious contracts disguised as fee tokens.
  - Can front-run, back-run, or sandwich attack message execution.
  - Can leave un-executed messages without expiration at destination which could be used to execute an exploit in the future or trigger a DoS under certain conditions.

Structural Design of the Teleporter Messenger

By design, the `TeleporterMessenger` contract will be deployed at the same address on all subnets. This simplifies the implementation by allowing an explicit assumption that the `TeleporterMessenger` will be deployed at the given address on a destination chain when sending a cross-chain message and reduces the likelihood of passing incorrect inputs.

To do so, Nick's deployment method will be used by broadcasting the same signed transaction over each chain that wants to implement the message bridge feature. This allows for trustless deployment whereby no single entity is responsible for deploying the contract. Further, the `TeleporterMessenger` contract itself does not contain any privileged functions and is not

intended to be upgradeable. While this reduces the attack surface, it does limit the possible ways to mitigate unexpected situations that could arise. One such scenario could be not being able to link the `TeleporterMessenger` contract to a new `ReceiptQueue` contract if the previous one became stuck.

Moreover, as chains might have different opcodes, gas costs, economic rules, and address aliasing, the deployment might require changes at the VM level to properly handle and deploy the contract at the desired address.

It is also worth mentioning that even though the `TeleporterMessenger` is meant to interact only with CCA contracts, any user can initiate a message at the origin, exposing the attack surface.

Integration Considerations for Cross-Chain Applications

Throughout the audit and while reviewing the example cross-chain applications, several important considerations became apparent. These are important for developers who wish to integrate their application with the `TeleporterMessenger` contract. Below is a non-comprehensive list of considerations for CCAs sending messages using the `TeleporterMessenger` contract and receiving messages through the use of the `ITeleporterReceiver` interface.

Contracts Integrating With the `TeleporterMessenger` Contract to Send Cross-Chain Messages

- Ensure the contract does not include any functionality for making arbitrary external calls. Otherwise, a user could send a malicious message from this contract to the `TeleporterMessenger` contract.
- Messages are not necessarily executed in the order they were submitted from the source chain. Consider potential implications.
- Consider the implications of `originChainID` being the current chain.
- The fee token used to pay the relayer is transferred to the `TeleporterMessenger` instance, ensure the appropriate approval is set to the exact amount transferred, and avoid unlimited approval.
- Consider what could happen if one of the blockchains becomes inaccessible.

- If users are able to specify the fee token for paying the relayer. Ensure the message cannot impact other users.
    - For example, if an ERC-20 bridge contains a function for creating the bridged token on the destination chain, and a malicious user calls this function with a malicious (or non-existent) ERC-20 token as the fee, no relayer will relay the message. As a result, the token may never be able to be created (implementation-specific) and might also never be able to be bridged as a result.

Contracts Implementing the `ITeleporterReceiver` Interface

- Validate that `msg.sender` is the `TeleporterMessenger` contract address.
- Avoid using `tx.origin` anywhere within the call chain. Even if the relayed message contains an `allowedRelayerAddresses` list, this does not explicitly prevent addresses not on this list from executing the message. If message execution initially fails, the message can be retried by anyone.
- Validate the `originSenderAddress` if necessary.
- Validate the `originChainID` if necessary.
- Consider the implications of `originChainID` being the current chain.
- Messages are not necessarily executed in the order they were submitted from the source chain. Consider potential implications.
- Consider the risks of a relayer re-ordering messages or front/back-running or sandwich attacking the execution of a message.
- Determine if a relayer could force execution to fail. This would be for the benefit of the relayer as it would still receive its fee but pay less for gas.

Update: *The AVA Labs team provided 5 pull requests addressing the issues presenting in this report. Each issue has been updated to reference the relevant pull request that addressed the issue. Fixes for some issues were not present in the provided pull requests and their fixes were found by using git blame against the `main` branch. Overall, there were significant changes to the codebase between the audited commit and the most recent commit. While reviewing fixes to issues presented in this report only changes related to the specific issue were reviewed and any unrelated modifications to the codebase were not reviewed. Thus, we strongly recommend the codebase undergo a future code review that includes all changes to mitigate the risk of potential vulnerabilities resulting from the out-of-scope changes.*

Medium Severity

M-01 Fees Do Not Follow a Predictable Calculation

When sending a message with the `TeleporterMessenger` contract, there is no check to ensure that the fee deposited for the relayer who will submit the message on the destination subnet will be sufficient to cover the gas cost incurred by executing the message. This could cause users to spend more than what is required to execute their message, or less, in which case the message may not be relayed as it would not be profitable for a relayer.

Some message bridging protocols remedy this by predicting the gas cost based on the size of the message and the expected gas limit on the other subnet, which is translated into `feeAsset` units. Moreover, such protocols provide a view-only function to predict this value off-chain before submitting the message.

Consider adding functionality to estimate the necessary fee and assert it against the transferred fee assets.

Update: *Acknowledged, not resolved. AVA Labs stated the following about the issue:*

> *The fee asset and amount "accepted" by a given relayer is defined by that individual relayer themselves, so it makes sense for those fee estimates to be published off-chain, out of the scope of the `Teleporter` contract. For instance, some dApps/subnets may run a relayer that relays messages for free to attract more users. In other cases, one relayer may only accept fees paid in AVAX, while another accepts fees paid in USDC or other arbitrary ERC20 tokens. The fee amount of a given asset expected for a given relayer may also depend on the current gas price at the destination chain for a given message, the current price of the native gas token being spent by the relayer, and the current price of the asset being used to pay the relayer's fee. This information isn't necessarily available on-chain, and even if it were, different relayers may use different sources for this information. For these reasons, we think it's best to leave fee estimation and publishing to be handled on a case-by-case basis off-chain.*

M-02 Nick's Method for Deployments Is Error-Prone

The `TeleporterMessenger` contract will be deployed at the same address on any subnet that wishes to allow cross-chain messaging using the Teleporter protocol. It is intended to use Nick's method to ensure that the contract is deployed at the same address on each subnet. Since Nick's method uses a signed transaction from a one-time address, a few concerns arise that could prevent deployment as intended. In particular:

- If the deployment fails, the signed transaction will not be able to be retried as the signed nonce in the transaction for the one-time address will be used. This could occur if the call to `WARP_MESSENGER.getBlockchainID()` fails.
- If a subnet defines non-standard costs for opcodes, the `gasLimit` in the signed transaction may not be large enough for the deployment transaction to be executed.
- The `gasPrice` must also be set within the signed transaction. The `gasPrice` can vary widely between subnets due to the popularity of the chain as well as the liquid supply of native tokens used for paying for gas. If a subnet has a low total supply of the native token relative to other chains, the specified `gasPrice` may not be large enough to get the deployment transaction included in a block. Conversely, if a subnet has a relatively large liquid supply of native tokens, the `gasPrice` may require spending a large quantity of native tokens. This could result in an expensive deployment transaction when measured in fiat currency.

Consider using an alternative to Nick's method such as `CREATE2`. Alternatively, consider ensuring that there is a fallback approach to deploying the `TeleporterMessenger` contract at the intended address and that its procedure is well documented. This is so that the actors involved in the deployment of a new subnet can mitigate this problem without any friction.

Update: *Partially resolved in pull request 64 at commit `5f8b039`. `blockchainID` assignment has been moved from the `constructor` function into the `receiveCrossChainMessage` function. However, it would preferable to use a one-time only dedicated method to initialize the variable to reduce the attack surface and the gas cost when receiving messages from other subnets. Moreover, the deployment documentation does not include the additional information to deploy the contract under problematic situations. AVA Labs stated the following regarding the issue:*

> *We have updated the `TeleporterMessenger` contract such that it has an empty/default constructor in order to minimize the potential for the deployment transaction to fail. Using `CREATE2` is not really a viable alternative to Nick's method. It requires that a factory contract is deployed at the same address on each chain prior to creating the*

> *`TeleporterMessenger` instance using the factory. In order to deploy the factory contract to the same address on each chain (without having a shared account key used by every chain, as required in this case), Nick's method would still need to be used in the same manner, with the same possible error cases for that transaction to fail. The fallback approach in mind for a scenario where the normal `TeleporterMessenger` deployment transaction using Nick's method fails for a subnet is for that subnet to specify the contract in their genesis for new subnets, or add the code in a state upgrade as part of a required network update for existing subnets. While more cumbersome than sending a pre-defined transaction, each allows a surefire fallback approach for getting the contract code the necessary address on a given chain.*

Low Severity

L-01 Abstract Contracts Allow Direct Modification of State Variables

In order to prevent re-entrant calls, the `_sendEntered` and `_receiveEntered` state variables in the abstract `ReentrancyGuards` contract track whether a call has been made to a "sender" or "receiver" function. Since these variables use `internal` visibility within an `abstract` contract, a child contract would be able to directly modify these values which could unintentionally allow a re-entrant call.

Consider using `private` visibility for all non-constant and non-immutable state variables in `abstract` contracts as OpenZeppelin's `ReentrancyGuard` contract does. If child contracts should be able to update values for the state variables, consider creating `internal` functions for updating these variables which emit appropriate events and verify that the desired conditions are met.

Update: *Resolved in pull request #106 at commit 43d4c728.*

L-02 Missing and Incomplete Docstrings

Throughout the codebase, there are several parts with missing or incomplete docstrings. In particular, function arguments and return values are consistently undocumented which limits

users' ability to determine appropriate input values and expected outputs. Further, the following parts are missing docstrings:

- Line 27 in `TeleporterMessenger.sol`
- Line 32 in `TeleporterMessenger.sol`
- Line 8 in `ITeleporterMessenger.sol`
- Line 13 in `ITeleporterMessenger.sol`
- Line 32 in `ITeleporterMessenger.sol`
- Line 15 in `ReceiptQueue.sol`
- Line 16 in `ReceiptQueue.sol`
- Line 17 in `ReceiptQueue.sol`
- Line 18 in `ReceiptQueue.sol`
- Line 8 in `IWarpMessenger.sol`
- Line 16 in `IWarpMessenger.sol`
- Line 27 in `SafeERC20TransferFrom.sol`

Consider thoroughly documenting all functions (and their parameters) that are part of any contract's public API. Functions implementing sensitive functionality, even if not public, should be clearly documented. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Partially resolved pull request #106 at commit 43d4c728. The following instances still remain undocumented:*

- *Line 37 in `TeleporterMessenger.sol`*
- *Line 8 in `IWarpMessenger.sol`*
- *Line 14 in `IWarpMessenger.sol`*

L-03 External Call Is a Return Bomb Vector

When a relayer executes a message, an external call is made to the `destinationAddress` specified by the message sender where the amount of gas the call can use is specified in the message by the `requiredGasLimit` value. The return data from this call is not used and only the `success` of the call is recorded. Although a gas limit is set for the call, the external contract can force the transaction to use gas in excess of the specified limit due to an open issue with the Solidity compiler. This is possible because the `TeleporterMessenger` contract pays for memory expansion to accommodate the return data, but the external contract controls the size of that return data.

Consider using assembly to explicitly prevent any memory from being allocated for the return data from the external call.

Update: *Resolved in pull request #119 at commit 1de7667. However, no test case for such an exploit has been added to the test suite. Consider including a test for this kind of attack.*

L-04 Multiple Mechanisms to Relay Receipts

When a message is received on a destination chain, a receipt must be sent back to the origin chain so that the relayer is able to claim its fee token for executing the message. Within the `TeleporterMessenger` contract, there are two different paths for a receipt to return to the origin chain. One is by the regular inclusion of the receipt in a message going back to the origin, which goes through the queue/dequeue process of the `ReceiptQueue` contract. The other is by a relayer calling the `retryReceipts` function and relaying only the specified receipt. The latter approach bypasses the `ReceiptQueue` contract, leaving in the queue receipts which have already been relayed. The different approaches are not only more difficult to follow and debug in case of exploits but also increase the attack surface and the overall gas consumption.

Consider unifying the receipt mechanism to reduce the attack surface and the complexity involved while still maintaining the possibility of moving stuck receipts forward.

Update: *Acknowledged, not resolved. The Ava Labs team stated:*

> *The secondary mechanism to allow for relaying specific receipts (now called "`sendSpecifiedReceipts`") was added per the recommendation of a previous audit, which had called out that the relayers may not be able to efficiently receive rewards in the case that messages flow predominantly in a single direction between two Teleporter instances, such that the receipt queue grows in the length over time. Using "`sendSpecifiedReceipts`", relayers are always able to redeem their reward themselves by sending a single empty message back to the destination. This would only be needed in the case that the receipt queue for the given chain has grown large and the relayer is not willing to wait for it to empty out in order to be able to redeem their rewards.*
>
> *Making such that receipts sent via "`sendSpecifiedReceipts`" are removed from the receipt queue (if they are still in the queue) would require additional indices on the queue to be able to look up specific receipts at arbitrary positions, and remove items from arbitrary positions in the queue. This would significantly increase the complexity of the queue implementation and the gas cost of the common path enqueue/dequeue calls*

> *for receipt operations. Given that a receipt being sent multiple times is not a correctness issue, it is preferred to keep the complexity and gas cost of expected cases minimized.*

L-05 Lack of Validation That Contract Receiving Messages Supports `ITeleporterReceiver` Interface

A contract that would like to receive messages from the `TeleporterMessenger` contract must implement the `receiveTeleporterMessage` function from the `ITeleporterReceiver` interface. This function is called when the message is being executed. If a contract does not implement the `receiveTeleporterMessage` function, the call will fail and be stored for a future execution.

However, if the receiving contract does not implement this function and instead implements a `fallback`, the call could be successful. This opens up the possibility whereby even though the message is being forwarded to the contract, the destination contract does not handle it but there is no reversion of the transaction. Moreover, implementing the `receiveTeleporterMessage` function cannot be used to assert that the destination address will be able to handle the message, leaving open the possibility of setting a destination address by mistake that will call its fallback function.

In favor of restricting the possibility of a message being delivered into a non-compatible contract on the destination subnet, consider using an EIP meant for introspection, such as EIP-1820, instead of assuming that contracts that do not possess a `receiveTeleporterMessage` hook cannot be called.

Update: *Acknowledged, not resolved. AVA Labs stated the following regarding the issue:*

> *We think that trying to determine if a given contract address implements a specific interface would introduce more complexities than the benefits it provides. Even if the contract implements the required interface properly, it still may be unable to handle messages properly due to other implementation issues, as noted in the report, so even the best of guard rails doesn't ensure the correctness of applications built to use Teleporter. It is the responsibility of applications using Teleporter to ensure that they are able to send and receive their specific messages properly.*

L-06 Message Fee Can Be Increased by Any Account

The `TeleporterMessenger` contract allows users to add additional fees to a submitted message on the origin chain. However, there is no restriction on who can add the fees to a certain message. This results in two possible problems:

- If the original sender wants to add additional fees to their own message, they may mistakenly send the fees to a different message as there is no validation to ensure that the `msg.sender` is the same one as the one that originally sent the message.
- Malicious users might prevent a message from being relayed by adding stolen or OFAC-sanctioned funds into a particular message. Relayers may not want to risk receiving tainted funds even if the original fees deposited came from an honest source. As a result, relayers may not relay such messages.

In order to prevent unauthorized deposits into a particular message and to reduce the possibility of mistakenly depositing fees into a different message, consider restricting the deposit to the one that originally sent the message. Alternatively, consider using an allow list of addresses that can add more funds, similar to how authorized relayers are handled on the destination chain.

Update: *Acknowledged, not resolved. The Ava Labs team stated:*

> *It is not practical to limit the ability to add fees for a message to the sender of the message given that in most cases the sender of the message will be smart contract accounts. Thus, in order to add fee amounts from the same account, the smart contract would need to implement further logic to allow it to call the* `TeleporterMessenger.addFeeAmount` *function on behalf of users, increasing complexity, gas cost, and the likelihood of error that leads to it being impossible for anyone to add fee amounts for a given message.*

L-07 Using Custom Ownership Logic

The `enqueue` and `dequeue` functions in the `ReceiptQueue` contract implement custom logic to ensure that only the owner of the contract (i.e., the `TeleporterMessenger` contract) is able to update the queue. Custom access control implementations are not recommended as they could introduce a larger attack surface if not implemented properly. Consider using OpenZeppelin's `Ownable` library instead.

Update: *Resolved in pull request 32. The* `ReceiptQueue` *contract has been adapted to be a library and no longer needs access control from those methods.*

L-08 Semantic Overload

Throughout the codebase, there are multiple instances where variables are used for multiple different purposes (Semantic-overloading. For instance:

- The `outstandingReceiptForDestination` variable keeps track of the `ReceiptQueue` contract instance for a specific destination chain ID. However, it is also used to check if such a contract was deployed earlier in a previous operation.
- The `relayerRewardAddress` address is meant to indicate the fee payment receiver after the message was sent. However, it is also used to flag whether a message has been delivered or not, even though fees might not be involved during the operation.
- The `getVerifiedWarpMessage` function from the `IWarpMessenger` interface will return the message when the `valid` flag is `true`, but nothing when it is `false`. Meaning that the actual `message` output has the same behavior as the flag instead of returning the possibility of a `false` outcome while returning the `message`. The same applies to the `getVerifiedWarpBlockHash` function.

This can lead to the codebase not only being harder to understand but also harder to reason about from a security point of view.

Consider using separate flags to store different states instead of reusing variables. This will help increase the readability and understandability of the codebase.

Update: *Partially resolved in pull requests 32 and 106. The* `outstandingReceipts` *mapping has been removed in favor of using the* `ReceiptQueue` *as a library. However, the* `relayerRewardAddresses` *is still being used to track if the message was received or not. AVA Labs stated the following regarding the issue:*

> `outstandingReceiptForDestination` *no longer exists given that the receipt queues are now implemented in a library. In the case of* `relayerRewardAddress`*, using it to track whether a message has been delivered results in less state used by the contract, and fewer places in state that need to be properly updated for each message received, reducing the overall code complexity and gas costs. We created an internal* `messageReceived` *helper function to help increase code clarity.*

L-09 Incentive Misalignment When Sending a Message

The `TeleporterMessenger` contract implements functionality to allow certain actors (called relayers) to relay messages for users in exchange for a fee. Once the relayer has fulfilled its part and delivered the message on the destination chain, a receipt is created on the destination chain to be used as verification. It signifies that the work was done and allows the relayer to collect their fee on the origin chain.

However, the procedure to send these possible third-party receipts back to the origin involves attaching them to a message going in the opposite direction. Even though there is a limit on the maximum number of receipts attached to a message (currently 5), users will have to pay for gas expenditures unrelated to their message.

Moreover, when a user is acting as a relayer by both sending and relaying their own message, the associated fee for the message can be zero, as they would simply be paying themselves. However, the user is still required to attach receipts from the origin chain and process them on the destination chain without any additional reward.

Even though there is a mechanism for relayers to relay specific receipts without an associated message using the `retryReceipts` function, users cannot opt out of attaching receipts using the `sendCrossChainMessage` function if there are receipts to be forwarded.

Consider adjusting the incentive design around the receipts mechanism and allowing users to define the number of receipts they want to attach to their message.

Update: *Acknowledged, not resolved. AVA Labs stated the following regarding the issue:*

> *Acknowledged. Allowing users to define the number of receipts they want to attach to their message would result in messages not containing any receipts except for those beneficial to the sender themselves. In this case, relayers would always have to manually send messages back to the origin chain themselves for their receipts to be returned, multiplying the message overhead and also increasing the total relayer cost to redeem rewards.*
>
> *While possibly suboptimal, we disagree that incentives are misaligned. Relayers must account for needing to provide gas for up to a maximum of 5 receipts being marked in state per message, and can do so by calculating that cost in order to determine if they are properly incentivized if charging per message. Message senders must account for this relayer cost in the fee to incentivize a relayer. Message senders relaying their own message (i.e., with no message fee), do have the additional gas cost of marking*

> *unrelated receipts, but this is considered a small cost overhead of using the Teleporter protocol.*

L-10 Relayer Rewards Might Get Stuck

When a relayer relays a message on the destination chain, it must pass a `relayerRewardAddress` address that can be different from the address that makes that call (`msg.sender`). Once the receipt for such a message submission is received on the origin chain, the implementation marks that fee as able to be withdrawn for the `relayerRewardAddress` address. The `relayerRewardAddress` address is then able to redeem the balance accumulated to it by calling the `redeemRelayerRewards` function.

However, if there is a contract deployed at the `relayerRewardAddress` address that does not implement functionality to call the `redeemRelayerRewards` function (for instance, if it is meant to be used as a cold storage wallet), the rewards will be stuck as there is no way to redeem rewards on behalf of another address.

Consider allowing anyone to call the `redeemRelayerRewards` function on behalf of an input `relayerRewardAddress` address. This will help prevent funds from becoming stuck for addresses that lack the functionality to call the `redeemRelayerRewards` function.

Update: *Acknowledged, not resolved. AVA Labs stated the following regarding the issue:*

> *Allowing anyone to call `redeemRelayerRewards` on behalf of an input `relayerRewardAddress` has possible unintended side effects. For instance, there may be tax implications around when rewards are redeemed, or certain relayer operators may not be allowed to redeem specific rewards due to regulatory considerations. Any relayer can set arbitrary reward addresses owned by others when delivering a message, so it may be undesirable to also allow anyone to redeem those rewards on their behalf.*

L-11 Relaying Message Without a Fee Requires Reward Address

When a message is submitted to the `TeleporterMessenger` contract, a fee is generally included that will be redeemable on the source chain by the relayer who executes the message on the destination chain. If a user wishes to relay their own message, the fee amount can be set to zero as they do not need to incentivize an external relayer to execute the message. When a user relays their message on the destination chain by calling the `receiveCrossChainMessage` function, they are expected to pass a non-zero

`relayerRewardAddress` address regardless of whether there was a fee associated with the message. This could be counter-intuitive for the user as they may omit the `relayerRewardAddress` parameter or pass the zero address as there is no relayer to pay a reward to.

In order to improve the readability of the codebase and prevent debugging possible reverted transactions, consider using a flag to mark that such state has been reached. Alternatively, consider using an `enum` containing all the possible states of the message.

Update: *Acknowledged, not resolved. AVA Labs stated the following regarding the issue:*

> *This requirement is documented in the code, and if a given message has no fee or the deliverer otherwise does not want to redeem the reward for it, they can provide a non-zero "black hole" address such as 0x00...01. While using separate variables to track the state of message delivery may help code readability, it comes at the trade-off of complexity and gas cost.*

Notes & Additional Information

N-01 Lack of Security Contact

Providing a specific security contact (such as an email or ENS name) within a smart contract significantly simplifies the process for individuals to communicate if they identify a vulnerability in the code. This practice proves beneficial as it permits the project owners to dictate the communication channel for vulnerability disclosure, eliminating the risk of miscommunication or failure to report due to a lack of knowledge on how to do so. In addition, if the contract incorporates third-party libraries and a bug surfaces in these, it becomes easier for the maintainers of those libraries to contact the appropriate person about the problem and provide mitigation instructions.

Throughout the codebase, there are contracts that do not have a security contact. For instance:

- The `ITeleporterMessenger` interface
- The `ReentrancyGuards` contract
- The `ITeleporterReceiver` interface

- The `TeleporterMessenger` contract
- The `SafeERC20TransferFrom` contract
- The `ReceiptQueue` contract
- The `IWarpMessenger` interface

Consider adding a NatSpec comment containing a security contact on top of the contract definitions. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and the ethereum-lists.

Update: *Partially resolved in pull request #106 at commit 43d4c728. The `IWarpMessenger` interface has not been updated to include a security contact.*

N-02 Unused State Variables

Within the `TeleporterMessenger` contract, there are multiple unused state variables. For instance:

- The `REQUIRED_ORIGIN_CHAIN_ID_START_INDEX` state variable
- The `MINIMUM_REQUIRED_CALL_DATA_LENGTH` state variable

To improve the overall clarity, intentionality, and readability of the codebase, consider removing any unused state variables.

Update: *Resolved in pull request #106 at commit 43d4c728.*

N-03 Constant Not Using `UPPER_CASE` Format

In `TeleporterMessenger.sol`, the `blockchainID` constant is not declared using `UPPER_CASE` format.

According to the Solidity Style Guide, constants should be named with all capital letters with underscores separating words. For better readability, consider following this convention.

Update: *Resolved in pull request #64 at commit 5f394a11. The `blockchainID` state variable is no longer defined as immutable and is assigned outside of the constructor.*

N-04 Multiple Instances of Missing Named Parameters in Mappings

Since Solidity 0.8.18, developers can utilize named parameters in mappings. This means mappings can take the form of `mapping(KeyType KeyName? => ValueType ValueName?)`. This updated syntax provides a more transparent representation of the mapping's purpose.

Throughout the codebase, there are multiple mappings without named parameters. For instance:

- The `queue` state variable in the `ReceiptQueue` contract
- The `latestMessageIDs` state variable in the `TeleporterMessenger` contract
- The `outstandingReceipts` state variable in the `TeleporterMessenger` contract
- The `sentMessageInfo` state variable in the `TeleporterMessenger` contract
- The `relayerRewardAddresses` state variable in the `TeleporterMessenger` contract
- The `receivedFailedMessageHashes` state variable in the `TeleporterMessenger` contract
- The `relayerRewardAmounts` state variable in the `TeleporterMessenger` contract

Consider adding named parameters to the mappings to improve the readability and maintainability of the codebase.

Update: *Resolved at commit `c2f43256`.*

N-05 Unused Named Return Variables

Named return variables are a way to declare variables that are meant to be used within a function's body for the purpose of being returned as the function's output. They are an alternative to explicit in-line `return` statements.

In `TeleporterMessenger.sol`, there are multiple instances of unused named return variables. For instance:

- The `messageID` return variable in the `sendCrossChainMessage` function
- The `messageHash` return variable in the `getMessageHash` function
- The `delivered` return variable in the `messageReceived` function
- The `relayerRewardAddress` return variable in the `getRelayerRewardAddress` function

- The `feeAsset` return variable in the `getFeeInfo` function
- The `feeAmount` return variable in the `getFeeInfo` function
- The `messageID` return variable in the `getNextMessageID` function
- The `messageID` return variable in the `_getNextMessageID` function

Consider either using or removing any unused named return variables, while keeping the same convention in all cases for consistency.

Update: *Resolved in pull request #106 at commit 43d4c728.*

N-06 Non-Explicit Imports Are Used

The use of non-explicit imports in the codebase can decrease the code clarity and may create naming conflicts between locally defined and imported variables. This is particularly relevant when multiple contracts exist within the same Solidity files or when inheritance chains are long.

Throughout the codebase, global imports are being used. For instance:

- Line 8 of `ReceiptQueue.sol`
- Lines 8-9 of `SafeERC20TransferFrom.sol`
- Lines 8-15 of `TeleporterMessenger.sol`

Following the principle that clearer code is better code, consider using named import syntax (`import {A, B, C} from "X"`) to explicitly declare which contracts are being imported.

Update: *Resolved in pull request #106 at commit 43d4c728.*

N-07 Lack of Event Emission

Within the `TeleporterMessenger` contract, the `redeemRelayerRewards` function allows a relayer to redeem rewards they are owed for relaying messages. This function does not emit an event besides the `Transfer` event that ERC-20 tokens should emit.

Consider emitting an event to ensure accurate off-chain monitoring of reward withdrawals without relying on external ERC-20 token event emissions.

Update: *Resolved in pull request 53 at commit `874d6d3`.*

N-08 Misleading Error Name

The `addFeeAmount` function in the `TeleporterMessenger` contract will revert when there is no message for the input `messageID` and `destinationChainID`. This occurs when the message hash for the corresponding message in the `sentMessageInfo` mapping is identically zero and will raise the `MessageAlreadyDelivered` error. The `MessageAlreadyDelivered` name for the error implies that it will be raised when a message has already been delivered when in fact it can also occur for invalid `destinationChainID`/`messageID` pairs.

Consider renaming this error message to ensure it is clear under what scenarios it may cause execution to revert.

Update: *Resolved in pull request 59.*

N-09 Redundant Getter Functions

In the `TeleporterMessenger` contract, there are `external` getter methods that are redundant given the `public` state variables. In particular:

- The `getRelayerRewardAddress` function is redundant given the `relayerRewardAddresses` mapping.
- The `checkRelayerRewardAmount` function is redundant given the `relayerRewardAmounts` mapping.

Consider either removing the aforementioned functions or reducing the visibility of the `relayerRewardAddresses` and `relayerRewardAmounts` state variables to `internal` or `private`.

Update: *Resolved in pull request #106 at commit 43d4c728. The visibility of the `relayerRewardAddresses` and `relayerRewardAmounts` mappings have been changed to `internal`.*

N-10 Typographical Errors

Consider correcting the following typographical errors to improve the readability of the codebase:

- On line 102 of `ITeleporterMessenger.sol`, "it's" should be "its".
- On line 142 of `TeleporterMessenger.sol`, "mush" should be "must".

- On line 350 of `TeleporterMessenger.sol`, "subsquent" should be "subsequent".
- On line 386 of `TeleporterMessenger.sol`, "preivously" should be "previously".
- On line 413 of `TeleporterMessenger.sol`, "exeuction" should be "execution".
- On line 648 of `TeleporterMessenger.sol`, "adress" should be "address".

Update: *Resolved in pull request #106 at commit 43d4c728.*

N-11 Interface Does Not Fully Represent the Implementation

In the `ITeleporterMessenger` interface, there are several getters and functions from the `TeleporterMessenger` contract, which implements `ITeleporterMessenger` interface, that are not declared in the interface.

In order to improve the readability and facilitate such methods for third-party developers using the protocol, consider adding those to the interface.

Update: *Partially resolved in pull request #106 at commit 43d4c728. Several getters remain that are not declared in the interface.*

N-12 Misleading Documentation

In the `SafeERC20TransferFrom` library, the docstring for the library states that it "checks the balance of the recipient". Since the recipient is hardcoded as `address(this)`, consider updating the docstring to make it clear that the recipient is always the contract using the library.

Update: *Resolved in pull request 106.*

N-13 Solidity Programming Best Practices and Recommendations

Throughout the codebase, several areas were identified where the code style could be improved. In particular:

- The `checkIsAllowedRelayer` function in the `TeleporterMessenger` contract could use either `internal` or `private` visibility. The function performs a simple computation that would likely be done off-chain by relayers and given that it is a `pure`

function, it has no dependence on the contract state. Moreover, the allowed relayer could simply pass the respective index at which their address is in the allow list instead of looping through all of it. Therefore, if it is not allowed, the transaction will revert as none of the possible provided indexes could pass the check.

- The events for the `TeleporterMessenger` contract are defined in the `ITeleporterMessenger` interface while the errors are defined within the `TeleporterMessenger` contract. Consider defining both the errors and events within the interface.
- Lines 782-787 of `TeleporterMessenger.sol` could be simplified by using a `min` function such as the one from the OpenZeppelin `Math` library.
- The `TeleporterMessageInput` struct is only used as an input to the `sendCrossChainMessage` function. Consider removing this redundant struct.
- The `outstandingReceiptForDestination` variable name does not align with its purpose which is to hold the `ReceiptQueue` contract for a given chain ID. Consider using a variable name that clearly indicates the purpose of the variable.

Consider addressing these areas to improve the readability of the codebase.

Update: *Partially resolved in pull request 106. Function visibility has been reduced and OpenZeppelin's `Math` library has been imported. AVA Labs stated the following regarding the issue:*

> *Still loop through allowed relayers where necessary to avoid extra parameters needing to be provided by the relayer, and considering an expected case is that the array is empty so that any relayer is allowed.*
>
> *Custom errors have been removed in favor of explicit require statements with custom string messages to allow for a more user-friendly error display in explorers.*
>
> *`TeleporterMessageInput` is now also used for `sendCrossChainMessage`, and is unfortunately necessary to reduce stack depth within the functions where it's used.*

N-14 Redundant Function Argument

The `feeContractAddress` argument for the `addFeeAmount` function in the `TeleporterMessenger` contract is redundant. This is because `contractAddress` is stored within the `feeInfo` struct, which is stored within the `sentMessageInfo` mapping.

Consider removing this argument to reduce redundant code.

Update: *Acknowledged, not resolved. The Ava Labs team stated:*

> *The redundancy `feeContractAddress` argument will be kept as a safeguard if callers specify the wrong message ID accidentally which could result in an unexpected asset being transferred out of their account. This case would be especially bad for assets with different "decimal" values where the amounts could be extremely high.*

N-15 Gas Optimizations

Possible gas cost improvements were found throughout the codebase. In particular:

- Within the `_sendTeleporterMessage` function in the `TeleporterMessenger` contract, storing the fee status when there is no fee included in the message costs unnecessary gas. Consider only storing the fee info when there is a non-zero fee included.
- Since the `TeleporterMessenger` contract will be deployed at the same address on all blockchains due to the use of Nick's method of deployment, a constant containing the contract address could be used in place of `address(this)` throughout the contract.
- Each time a message is sent to a new blockchain, a new instance of the `ReceiptQueue` contract is deployed. Consider using the minimal proxy pattern to reduce the cost for deploying instances of the `ReceiptQueue` contract, or even better, using the same `ReceiptQueue` contract for all the destination subnets and adding an extra layer in the `queue` mapping to select the particular queue for the destination chain ID.
- The `_getOutstandingReceiptsToSend` function in the `TeleporterMessenger` contract makes at most `MAXIMUM_RECEIPT_COUNT + 1` external calls to an instance of the `ReceiptQueue` contract. This could be reduced to a single external call by implementing a method within the `ReceiptQueue` contract that will return up to the input number of values from the front of the queue.
- The `ReceiptQueue` contract is only used in the context of the `TeleporterMessenger` contract, consider converting this contract to a library to eliminate expensive external calls for interacting with the queue.

To reduce gas consumption during the execution of the code, consider addressing these parts of the codebase.

Update: *Partially resolved in pull requests 106 and 32. The `ReceiptQueue` contract is now a library used inside the `TeleporterMessenger` contract. The `getOutstandingReceiptsToSend` function has been removed and its logic merged into the `ReceiptQueue` library. AVA Labs stated the following regarding the issue:*

> *To make any future changes less error-prone, `address(this)` will continue to be used.*
>
> *Still store the fee information to allow for the possibility of adding future fee amounts of the same provided asset contract.*

Recommendations

Monitoring Recommendations

While audits help in identifying code-level issues in the current implementation and potentially the code deployed in production, the AVA Labs team is encouraged to consider incorporating monitoring activities in the production environment. Continuous monitoring of deployed contracts helps identify potential threats and issues affecting production environments. With the goal of providing a complete security assessment, this section recommends several actions addressing trust assumptions and out-of-scope components that can benefit from on-chain monitoring.

Suspicious Activity

- OFAC-sanctioned addresses interact with your smart contract.
- Destination contracts on the destination subnet do not belong to any familiar protocol or project.
- Failed messages that are stored at the destination chain due to trying to be executed at addresses without any code (ticking bomb).
- Higher-than-average payload size.
- Validators' downtime for signing bundle messages.
- Calls that have the `TeleporterMessenger` contract as the recipient.

Financial

- Front-run scenarios on an ongoing message delivery.
- Users make use of a custom token to pay the fees.
- Receipts pile up in a particular chain and do not return to their origin chain.
- `TeleporterMessenger` contracts' fee balances.

Technical

- `ReceiptQueue` contract reaches a stuck scenario where it cannot retrieve the `size` output.

Conclusion

The `TeleporterMessenger` implementation provides a standardized way for users to send messages between subnets within the Avalanche network. It includes numerous checks necessary for cross-chain applications that want to send and receive cross-chain messages. The security of a cross-chain application that integrates with the `TeleporterMessenger` system relies on the proper integration and usage of the messaging system. As outlined in this report, there are several important considerations that CCAs need to consider to ensure their own security. Ultimately, these will be dependent upon the purpose of the CCA.

The codebase is generally robust and no critical or high severity issues were found during the audit. However, while reviewing the codebase, there were a few key areas that could be improved. Specifically, how the receipts are handled, the use of semantic overloading for state variables to track various states of the contract, the planned method of deployment across subnets and having a fail-safe method for non-standard subnets, along with the overall documentation.

Overall, the Ava Labs team was responsive to questions throughout the audit and also very helpful in guiding our understanding of the `TeleporterMessenger` contract. In addition, they provided guidance regarding external dependencies required for the cross-chain messaging system, such as how actors intervene in the Warp protocol, the message pipeline outside the Solidity contracts, and a high-level view of the rest of the Avalanche network project.