OpenZeppelin | security

Teleporter Token Bridge Audit

Ava Labs.

June 26, 2024

Table of Contents

Summary

Type	Bridge
Timeline	**(Phase 1)** From 2024-04-15
	To 2024-04-30
	(Phase 2) From 2024-06-03
	To 2024-06-07
Languages	Solidity

Total Issues	14 (11 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	1 (1 resolved)
Medium Severity Issues	2 (2 resolved)
Low Severity Issues	2 (1 resolved)
Notes & Additional Information	8 (6 resolved, 1 partially resolved)
Client-Reported Issues	1 (1 resolved)

Scope

The audit took place in two phases.

Phase 1

During phase 1, we audited the ava-labs/teleporter-token-bridge repository at commit f18d6f3.

In scope were the following files:

```
contracts/src
├── ERC20Destination.sol
├── ERC20Source.sol
├── NativeTokenDestination.sol
├── NativeTokenSource.sol
├── TeleporterTokenDestination.sol
├── TeleporterTokenSource.sol
├── interfaces
│   ├── IERC20Bridge.sol
│   ├── IERC20SendAndCallReceiver.sol
│   ├── INativeSendAndCallReceiver.sol
│   ├── INativeTokenBridge.sol
│   ├── INativeTokenDestination.sol
│   ├── ITeleporterTokenBridge.sol
│   └── IWrappedNativeToken.sol
└── utils
    ├── CallUtils.sol
    ├── SafeWrappedNativeTokenDeposit.sol
    └── SendReentrancyGuard.sol
```

Phase 2

During phase 2, we audited the ava-labs/teleporter-token-bridge repository at commit 8922c54.

In scope were the following files:

```
contracts/src
├── WrappedNativeToken.sol
├── interfaces
│   ├── IERC20SendAndCallReceiver.sol
│   ├── IERC20TokenBridge.sol
```

```
│   ├── INativeSendAndCallReceiver.sol
│   ├── INativeTokenBridge.sol
│   ├── ITokenBridge.sol
│   └── IWrappedNativeToken.sol
├── TokenHub
│   ├── interfaces
│   │   ├── IERC20TokenHub.sol
│   │   ├── INativeTokenHub.sol
│   │   └── ITokenHub.sol
│   ├── ERC20TokenHub.sol
│   ├── NativeTokenHub.sol
│   └── TokenHub.sol
├── TokenSpoke
│   ├── interfaces
│   │   ├── INativeTokenSpoke.sol
│   │   └── ITokenSpoke.sol
│   ├── ERC20TokenSpoke.sol
│   ├── NativeTokenSpoke.sol
│   └── TokenSpoke.sol
└── utils
    ├── CallUtils.sol
    ├── SafeWrappedNativeTokenDeposit.sol
    ├── SendReentrancyGuard.sol
    └── TokenScalingUtils.sol
```

System Overview

The Teleporter Token Bridge enables users to bridge tokens between different subnets in the Avalanche network by leveraging the Teleporter protocol which facilitates cross-subnet communication through Avalanche Warp Messaging (AWM). A bridge is capable of bridging a single ERC-20 token or native asset from a source chain, referred to as the 'hub', to one or more destination chains, referred to as 'spokes'. This bridged token can either be represented as an ERC-20 token or as the native asset on the destination chain.

Each bridge consists of a smart contract on the hub chain and a smart contract on each of the supported spoke chains. These smart contracts communicate via Teleporter messages and the contract on the hub chain tracks the amount of tokens bridged to each spoke chain. These amounts are temporarily locked in the contract on the hub chain acting as collateral for the representation of these tokens on the spoke chain. The tokens are only unlocked on the hub chain when they are returned from the spoke chain.

The bridge contract of the spoke chain needs to be registered on the bridge contract of the hub chain before funds can be transferred between the two chains. This is accomplished by sending a registration message from the spoke to the hub containing information such as the amount of decimals used on the spoke chain to allow for proper scaling of the tokens in case there is a difference in decimals between the hub and spoke chain. Registration of the spoke on the hub also enables the hub to protect users from bridging funds to non-existent spokes.

While bridging funds between the hub and spokes is bi-directional using a single bridge operation, the system also supports bridging tokens between different spokes on different chains. This `multihop` feature consists of two bridge operations, one from the sending spoke to the hub and one from the hub to the receiving spoke. In between these two bridge operations, the amount of tokens per spoke is updated on the hub.

On top of bridging funds between different subnets, the Teleporter Token Bridge also implements a `send and call` feature, allowing users to bridge funds and directly use them in a call to a smart contract on the destination chain. This facilitates more complex scenarios such as performing swaps or paying for service fees for protocols on the destination chain. In case this call fails, the bridged funds are sent to the specified recipient address on the receiving chain.

When bridging tokens from the hub to a spoke, the bridged tokens could be represented as native assets on the spoke chain. To support this, the spoke handles the minting of the native asset by interacting with a precompiled contract. Burning the native assets is achieved by locking the tokens in the spoke such that they are also unredeemable on the hub. Similarly, native assets burned as part of transaction fees are made unredeemable on the hub.

Upon creating a new subnet, the subnet can allocate an arbitrary amount of native assets in the genesis block. As such, the bridge on the spoke chain first needs to be bootstrapped before it can be used to permissionlessly bridge funds between the hub and spoke. In order to accomplish this, the deployer of the spoke has to send the same amount of tokens as the amount of native assets minted in the genesis block of the spoke chain to the hub. These tokens will be locked on the hub to act as collateral for the native assets minted in the genesis block of the spoke chain. From this moment, the bridge contract on the spoke chain is said to be collateralized and can be used to bridge funds bidirectionally between the hub and the spoke.

The actual transmission of the Teleporter messages between different chains is handled by off-chain relayers. In order to incentivize these relayers, each type of bridge message includes a configurable fee. The user can specify both the token in which the fee will be paid, as well as the fee amount itself. In case a user initiates a bridge message with a below-market fee, the Teleporter protocol allows the user to increase the fee, incentivizing relayers to deliver the bridge message to the destination chain.

Security Model and Trust Assumptions

The smart contracts forming the Teleporter Token Bridge are permissionless. As long as a compatible token bridge instance exists on the spoke chain, users can transfer tokens from the hub chain to that specific spoke chain. However, each smart contract requires precise configuration during deployment. For contracts on the spoke chain inheriting from `TokenSpoke`, proper settings for the values in `TokenSpokeSettings` are required to successfully register the spoke on the hub.

The security of the Teleporter Token Bridge contracts heavily depends on the Teleporter Protocol. This setup presumes the protocol functions as intended and adheres to its specified properties. In addition, it is crucial that the native minter precompile be configured to

exclusively allow the `NativeTokenSpoke` contract address to execute `mintNativeCoin`, preventing under-collateralization of the bridge.

Each subnet features unique modifications at the VM level, affecting gas costs, opcodes, native coins, and more, highlighting the importance of tailored settings for each environment. As such, users of the Teleporter Token Bridge should ensure that tokens are sent to addresses or contracts which are capable of receiving funds on the destination chain. If the call to the receiving contract of a `send and call` message fails, the funds are sent to a user-specified fallback recipient on the destination chain. However, in case this fallback recipient or the recipient of a normal `send` message is not able to receive the bridged tokens, it will result in a loss of funds, as the funds will remain locked in the bridge on the receiving chain.

High Severity

H-01 Teleporter Fee Is Minted Out of Thin Air - Phase 2

When a user wants to bridge tokens from a `TokenSpoke` on chain B back to a `TokenHub` on chain A, they have to pay a Teleporter message fee. This fee is used to incentivize relayers for passing the message from chain B to chain A. Users are free to choose both the fee amount and the token used as the fee by specifying the `primaryFeeTokenAddress` and `primaryFee` fields of the `SendTokensInput` struct passed to the `send` or `sendAndCall` function. The Teleporter only supports fees being paid in ERC-20 tokens. Thus, if a user wants to pay in native tokens, they have to wrap the native tokens into wrapped native tokens.

After calling `send` or `sendAndCall` with the desired fee token and amount to pay for relaying the Teleporter message, the `_prepareSend` function will call `_handleFees` in order to ensure that the fee is transferred from the sender to the `TokenSpoke` contract. However, in the particular case where the `TokenSpoke` contract is a `NativeTokenSpoke` and the user wishes to pay the Teleporter fee in wrapped native tokens, the `_handleFees` function will call `_deposit` of `NativeTokenSpoke`. This function will mint wrapped native tokens out of thin air instead of transferring the wrapped native tokens from the sender to the `NativeTokenSpoke` contract. Users might specify an arbitrarily high fee amount, which will dilute the value of wrapped native tokens as they will lose their peg with the native tokens on chain B. Additionally, the user could opt to relay the message themselves in order to claim this arbitrarily high amount of wrapped native tokens.

Instead of minting the fee amount out of thin air, consider ensuring that the specified fee amount is sent from the user to the `NativeTokenSpoke` contract before bridging the tokens.

Update: *Resolved in pull request #158. If the user wishes to pay the fee in wrapped native tokens, the `_handleFees` function will now first approve the wrapped native tokens from the sender and afterwards pull the wrapped native tokens from the sender into the `NativeTokenSpoke` contract.*

Medium Severity

M-01 Bridging Funds Smaller Than the Minimum Denomination Results in Loss of Funds - Phase 1

When sending tokens from the source chain to the destination chain, the received tokens are scaled in the destination bridge. Depending on the configured `decimalsShift` and `tokenMultiplier` of the `TeleporterTokenDestination` contract, the received tokens might be scaled down in the `_receiveTeleporterMessage` function. If the amount of received tokens is smaller than the `tokenMultiplier`, the division in the `scaleTokens` function will result in 0 and thus no tokens will be minted on the destination chain.

This creates an imbalance between the funds locked in the bridge contract on the source chain and those on the destination chain. As a result, the tokens on the destination chain will become over-collateralized, and some tokens will remain locked in the source chain forever. Ideally, the bridge contract on the home chain should prevent users from bridging funds that are smaller than the minimum denomination of the destination chain. However, since the home chain is currently unaware of the decimal configuration on the destination chain, this limitation might necessitate a redesign of the current system.

Consider redesigning your system to enable the home chain to recognize the decimal settings of the destination chain. Alternatively, ensure that this behavior is well-documented to prevent user losses.

Update: *Partially resolved in pull request #108 and pull request #124. The Ava Labs team stated:*

> *We decided to add a required "destination registration" step for a source to learn of new destination contracts prior to allowing tokens to be bridged to them. In addition to preventing fund amounts smaller than the minimum denomination from being able to be bridge, it also (possibly more significantly) adds a guard rail to prevent against funds being bridged to invalid/non-existent destination contracts in the event a wrong blockchain ID or address is provided by mistake.*
>
> *The registration steps work by sending a Teleporter message from the destination contract to the source contract via calling `registerWithSource` on the destination contract. This message includes the destination's denomination information, as well as its initial reserve imbalance (used in the case of `NativeTokenDestination`). When*

> *this `RegisterDestinationMessage` is delivered to the source contract, it is added to the mapping of registered destinations. The source contract is now responsible for scaling the token amounts to be sent to destinations with different denominations and also collecting sufficient collateral to account for a destination's initial reserve imbalance prior to allowing funds to be bridged to that destination.*
>
> *Collateral can be added to the source contract for a specific destination by calling `addCollateral`. This is a UX improvement since users are now unable to call `send` for destinations that are not yet collateralized, which previously would add collateral but not result in any tokens being minted on the destination. In the case of multi-hop transfers, it is possible for the intermediate transaction on the source chain to fail in the event that the destination specified by the first message is not properly registered. To prevent funds from being locked in this case, we added a `multiHopFallback` recipient where the tokens are sent on the source chain in this event.*
>
> *Note that any address on any Subnet could register itself as a destination whether or not it is implemented properly. The fact that a destination is registered with a source contract that is known to be verified/correct does not mean that the destination contract is correct/trustworthy. While this design does add an additional step to the set-up of a new destination contract, we think the guard rails it provides makes it worth adding.*

The new design requires the destination bridge to register its token configuration on the source bridge by means of a registration message. Registering the token configuration on the source bridge enables bridges to scale tokens to the denomination of the receiving bridge before actually bridging the tokens. In the event that this scaling results in zero tokens, no tokens are bridged and the user will not lose any funds. While the implementation of this new design looks good, consider taking into account the following:

- *In the unlikely event that native assets are sent to `fallbackRecipient` or `multiHopFallback` and these contracts are not payable, the funds will remain locked in the bridge. Consider implementing a recovery mechanism which allows the original sender to retrieve the funds in case these fallback calls fail.*
- *The `bridgedBalances` mapping to keep track of the different bridged balances on the source chain stores the balance in the denomination of the destination chain. As the mapping is public, other contracts on the source chain may use this balance. We think it would be valuable to mention this somewhere in the docstrings of the mapping.*
- *The function `registerWithSource` allows users to specify the fee token and fee amount to incentivize the relayer to send cross-chain messages to the source chain. However, this specific fee token is not sent to the bridge instance. When sending a Teleporter message with `_sendTeleporterMessage`, the bridge will attempt to*

approve the `TeleporterMessenger` to use the fees, which will fail in case the bridge instance does not own those funds. Consider first sending the fees to the bridge before calling `_sendTeleporterMessage`.

***Update 2:** Resolved in pull request #139 and pull request #153. The above three items have been resolved as follows:*

- *A recovery mechanism has not been implemented. However, the documentation in `ITokenBridge` now properly states that both `fallbackRecipient` and `multiHopFallback` should be able to receive tokens.*
- *The documentation of the `bridgedBalances` mapping now states that the balances are represented in the destination token's denomination.*
- *The `registerWithSource` function now calls `_handleFees` which pulls the fees from the sender to the bridge before calling `_sendTeleporterMessage`.*

M-02 Inability to Identify Originating Transaction Address on Destination Chain - Phase 1

The Teleporter Token Bridge contracts enable the transfer of tokens between different subnets by specifying a recipient on the destination chain using the _send method. In addition, they also support token transfers using a function call with the _sendAndCall method. However, the context provided as input when executing these methods may not be sufficient for users as it does not specify the sender of the originating transaction.

This could impact the user experience and may limit the bridge's applications. Consider a scenario where a user wants to authenticate the original sender to restrict calls. They might configure the contract to be callable only by the bridge contract but may also need to identify who initiated the transaction on the source chain. This could be critical, especially if the contract handles value, as they would not want to receive calls from just any origin sender.

Consider including a field in the message received by the targeted smart contract that represents the origin sender's address.

***Update:** Partially resolved in pull request #101. The Ava Labs team stated:*

> *We added the verified `sourceBlockchainID` and `originSenderAddress` values to the `receiveTokens` interfaces for both ERC-20 tokens and the native token to allow for `sendAndCall` use cases that require authenticating the caller. In a multi-hop case, these values are passed through the source chain on their intended destination. This required adding the fields to the `SingleHopCallMessage` payload. The*

> *`MultiHopCallMessage` only needs to include the `originSenderAddress` because the `sourceBlockchainID` is the source blockchain ID of the Teleporter message itself.*

The new implementation properly passes the original caller and source blockchain ID to the recipients of the `sendAndCall` feature. This enables use cases that require authenticating the caller on the source blockchain who initiated the transaction. On top of passing the caller and the source blockchain ID, consider adding the source bridge address. This allows the recipient of the `sendAndCall` feature to ensure the `sendAndCall` was handled by a trusted source bridge contract.

***Update 2:** Resolved in pull request #136. The new implementation now also passes the `originBridgeAddress` to the `receiveTokens` interface in both the `IERC20SendAndCallReceiver` and `INativeSendAndCallReceiver`. This allows the recipient contracts of the `sendAndCall` feature to verify that the `sendAndCall` was handled by a trusted source bridge contract.*

Low Severity

L-01 `transfer` and `send` Calls Are No Longer Considered Best Practice - Phase 1

When `transfer` or `send` calls are used to transfer native assets to an address, they forward a limited amount of gas. Given that EVM operations are sometimes re-priced, code execution on the receiving end of these calls cannot be guaranteed in perpetuity.

Throughout the codebase, there are instances where `transfer` or `send` is used to transfer native assets:

- In line 252 of `NativeTokenDestination.sol`, native assets are transferred via `transfer`.
- In line 343 of `NativeTokenDestination.sol`, native assets are transferred via `transfer`.
- In line 387 of `NativeTokenDestination.sol`, native assets are transferred via `transfer`.
- In line 87 of `NativeTokenSource.sol`, native assets are transferred via `transfer`.
- In line 119 of `NativeTokenSource.sol`, native assets are transferred via `transfer`.

Instead of using `transfer` or `send`, consider using `address.call{value: amount}("")` or the `sendValue` function of the OpenZeppelin `Address` library to transfer native assets. As more gas is forwarded with transfers using this approach, reentrancy vectors might become possible. However, the system properly prevents malicious reentrancy using the `sendNonReentrant` modifier.

Update: *Resolved in pull request #104. The Ava Labs team stated:*

> *We updated the contracts to use the `sendValue` function from the OpenZeppelin Address library.*

L-02 Potential Misconfiguration of ERC-20 Token Decimals - Phase 2

The `ERC20TokenHub` constructor allows the deployer to specify both the ERC-20 token contract address and the number of decimals for the token. This approach introduces a risk of misconfiguration, as the manually provided decimals may not accurately reflect the token's actual decimals. To minimize configuration errors and ensure consistency, it is recommended to retrieve the token's decimals directly using the `decimals()` function from the ERC-20 token within the constructor. This method would reduce the potential for error in the configuration.

Consider obtaining the token decimals from the ERC-20 token instead of requiring it as an argument in the constructor.

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

> *Since `decimals()` is not part of the `IERC20` interface, we want to maintain the possibility for the bridge contracts to be used for ERC-20's that do not provide that function, which requires the decimals to be provided explicitly.*

Notes & Additional Information

N-01 Use Custom Errors - Phase 1

Since Solidity version 0.8.4, custom errors provide a cleaner and more cost-efficient way to explain to users why an operation failed. Throughout the codebase, instances of `revert` and `require` messages were found.

For conciseness and gas savings, consider replacing `require` and `revert` messages with custom errors.

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

> *We chose to keep the use of custom errors for now at the expense of having slightly higher gas costs because we have observed that explorers are able to better display the error messages when custom error strings are used with `require` statements. If this changes in the future, we can update the contracts and the new versions will still be compatible with prior versions.*

N-02 Functions Are Updating the State Without Event Emissions - Phase 1

The `_mintNativeCoin` function in `NativeTokenDestination.sol` is updating the state without an event emission.

Consider emitting events whenever there are state changes to make the codebase less error-prone and improve its readability.

Update: *Resolved. The Ava Labs team stated:*

> *We discussed this issue with the OpenZeppelin team and came to the agreement that the `NativeCoinMinted` event emitted by the `NativeMinter` precompile is sufficient for this case. This event definition can be seen here. We decided to not add any additional events because they would be functional duplicates of the one emitted by the precompiled contract.*

N-03 Misleading Comments - Phase 1

The following misleading and inconsistent comments have been identified in the codebase:

- According to this comment, the value in wei to send to the contract is zero. However, this `value` parameter can be any value and does not need to be 0.
- The `totalNativeAssetSupply` function could benefit from some additional documentation due to the fact that this might be an approximation of the total native asset supply on the destination chain.

Consider revising the comments to improve consistency and more accurately reflect the implemented logic.

Update: *Resolved in pull request #103.*

N-04 Implicit Cast in `deriveTokenMultiplierValues` - Phase 2

In the `TokenScalingUtils` contract, the `deriveTokenMultiplierValues` function performs an exponentiation on the result of the subtraction of two `uint8` values. Although it is not strictly necessary, it is good practice to avoid implicit casts.

Consider adding an explicit cast to the result of the subtraction to `uint256` before performing the exponentiation.

Update: *Resolved in pull request #163.*

N-05 Unnecessary Import in `ERC20TokenSpoke` - Phase 2

The `ERC20TokenSpoke` contract includes an import statement for the `SafeERC20` library and applies the `SafeERC20` library on instances of `IERC` by means of the `using` keyword. However, none of the functions of the `SafeERC20` library are actually used in the `ERC20TokenSpoke` contract. Unused imports can lead to confusion and reduce the overall clarity and readability of the codebase.

Consider removing this unused import to avoid confusion and improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #163.*

N-06 Misleading comments - Phase 2

The following misleading and inconsistent comments have been identified in the codebase:

- According to the documentation of the `SpokeBridgeSettings` struct, it holds the `tokenDecimals` value. However, this value is not present in the implementation of the struct.
- This documentation in the `TokenSpoke` contract still uses the destination/source terminology instead of the new spoke/hub terminology.

Consider revising the comments to improve consistency and more accurately reflect the implemented logic.

Update: *Resolved in pull request #163.*

N-07 Gas optimizations - Phase 2

A few places in the codebase could benefit from gas optimization, for example:

- When the user wishes to bridge native tokens using `NativeTokenHub`, the native tokens are wrapped into wrapped native tokens and held as collateral in the `NativeTokenHub` contract. Afterwards, once the `NativeTokenHub` receives a message from `TokenSpoke` to release the collateral, the wrapped native token is unwrapped to native tokens upon withdrawing in case of a simple `send` or upon calling the recipient in case of a `sendAndCall`. Instead of wrapping and unwrapping the native token, consider leaving the native token untouched in the contract. The only exception to this rule would be when handling the fees of `multihop` transfers. As Teleporter fees only support ERC-20 tokens and fees for `multihop` transfers are paid within `NativeTokenHub`, these native tokens should be wrapped into wrapped native tokens.

- When the user wishes to bridge native tokens using `NativeTokenSpoke`, the native tokens are wrapped into wrapped native tokens. Afterwards, both the wrapped native tokens as well as the native tokens itself are immediately burned as they will result in a release of collateral on the `TokenHub` side. Consider simplifying this process by leaving out the wrapping of the native tokens such that only the native tokens need to be burned.

Consider optimizing these code sections to make them more gas efficient when bridging funds.

Update: *Partially resolved in pull request #164. The Ava Labs team stated:*

> *We chose to not make further changes to the `NativeTokenHub` contract for the gas optimization proposed because it would require an additional special case for handling multi-hop messages on a `NativeTokenHub` compared to the abstract TokenHub, and we didn't want to introduce the additional complexity.*

While the gas optimization in `NativeTokenHub` is not addressed, the optimization in `NativeTokenSpoke` is correctly implemented. Now, the native tokens sent to the bridge contract are directly burned instead of first wrapping them into wrapped native tokens. In order to be compatible with the changes made in `TokenSpoke`, `ERC20TokenSpoke` now directly burns the bridged tokens from the user's address instead of transferring them into the bridge contract and burning them afterwards.

N-08 Inconsistent Usage of `msg.sender` - Phase 2

Throughout the codebase, the `_msgSender` function is used to determine the address that is calling a function. This is generally used for compatibility with ERC-2771, allowing for the use of meta-transactions. The following functions **leverage** the `safeTransferFrom` function from the `SafeERC20TransferFrom` library **to pull** tokens from the original sender into the bridge contracts:

- The `_prepareSend` function in `TokenHub` to pay for Teleporter fees on the hub chain
- The `_handleFees` function in `TokenSpoke` to pay for Teleporter fees on the spoke chain
- The `_deposit` function in `ERC20TokenHub` to transfer the amount to bridge from the hub chain to a spoke chain

However, as this `safeTransferFrom` function is using `msg.sender` instead of `_msgSender`, the tokens will be pulled from the trusted forwarder instead of the original transaction signer in case meta-transactions are used to interact with the bridge contracts and the `_msgSender` function is overwritten. This might make it possible for transaction signers to steal tokens from the trusted forwarders **if** the trusted forwarders do not properly validate the transactions before submitting them on-chain.

Consider using `_msgSender` within all contracts that are intended to be inherited elsewhere in the codebase to ensure that meta-transactions are universally supported.

Update: *Resolved in pull request #165. All instances of `safeTransferFrom` are now using `_msgSender` instead of `msg.sender`.*

Client Reported

CR-01 Incorrect `secondaryFee` Denomination Could Lead to User Losses - Phase 1

All messages sent by `TeleporterTokenDestination` instances are sent to the specified token source contract. Therefore, if the final destination is not the source, a `multihop` is performed and two bridge messages will be sent. The first bridge message will send the tokens from the sending destination bridge to the source bridge, while the second message will send the tokens from the source bridge to the receiving destination bridge. When initiating the first bridge message from the destination bridge, the user must specify two fees:

1. The `primaryFee`, which will be used to pay relayers to relay the first bridge message of a `multihop`
2. The `secondaryFee`, which will be used to pay relayers to relay the second bridge message of a `multihop`

Before sending the first bridge message, the sending destination bridge properly scales the amount of funds to the denomination used by the source bridge. However, the `secondaryFee` remains denominated in the sending destination's chain token. Next, the source chain will attempt to use this fee without any conversion in the second cross-chain message of the `multihop` to the final destination chain. Depending on whether the denomination on the sending destination bridge has a higher or lower amount of decimals than the source bridge, the following two scenarios could unfold:

1. The denomination on the sending destination bridge has a lower amount of decimals. Therefore, the fee is too low according to the source bridge and the relayer will not pick up the message. The user has to manually increase the fee of the message by interacting with the `TeleporterMessenger` in order to incentivize a relayer to deliver the message to the final destination bridge. While there is no loss of funds, this scenario is likely when there is a difference in token decimals.
2. The denomination on the sending destination bridge has a higher amount of decimals. As the sending-destination-bridge-denominated fee is subtracted from the source-bridge-denominated amount to send, this might either revert (if `amount <= fee`) or cause the user to pay a significant amount of the sent amount in fees (if `amount > fee`). Moreover, if the execution reverts on the source bridge, the funds will be locked on the sending destination bridge, and there will be no way to unlock the funds. Similar to

the other scenario, this is likely to occur when there is a difference in token decimals. On top of that, this scenario would result in a loss of funds for the user.

Update: *Resolved in pull request #109. When performing a `multihop`, the source bridge is now responsible for scaling both the received funds as well as the `secondaryFee` to its own denomination upon receiving the first bridge message from the sending destination bridge. Afterwards, this `secondaryFee`, denominated in the source bridge's token, is used to send the second cross-chain message of the `multihop` to the final destination chain.*

Conclusion

The Teleporter Token Bridge facilitates the bridging of funds across different Avalanche subnets by leveraging the Teleporter cross-chain messaging protocol.

In accordance with other projects created by the Ava Labs team, the codebase was found to be of very high quality, having extensive documentation and a comprehensive suite of tests that covers a large surface of the codebase. During the audit, which was split up in two phases, one high-severity and two-medium severity issues were identified in addition to several issues of lesser severity. On top of that, Ava Labs reported an additional issue between the two phases. Both the high-severity issue and medium-severity issues have been resolved.

Communication with the team has been excellent, with all questions from the auditors being answered promptly and comprehensively.