From: n-var CONSULTING

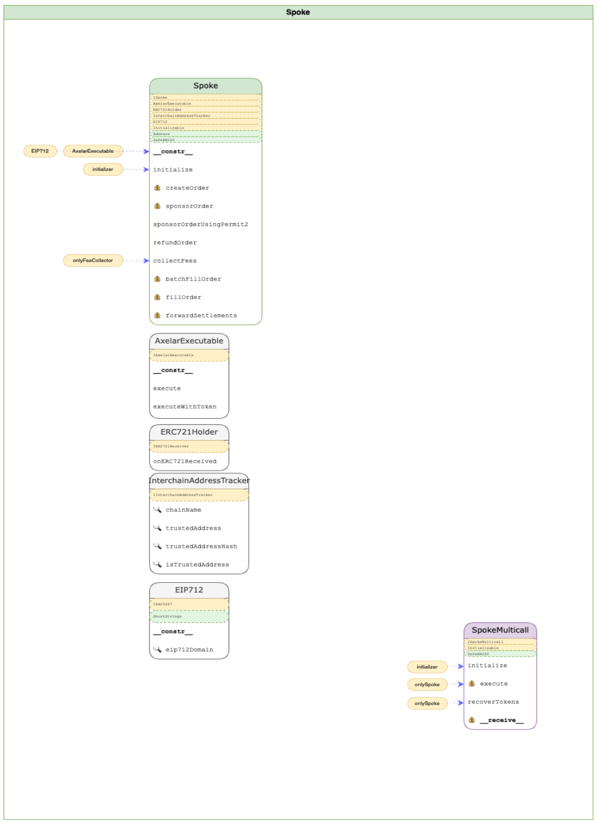
To: 0xSquid

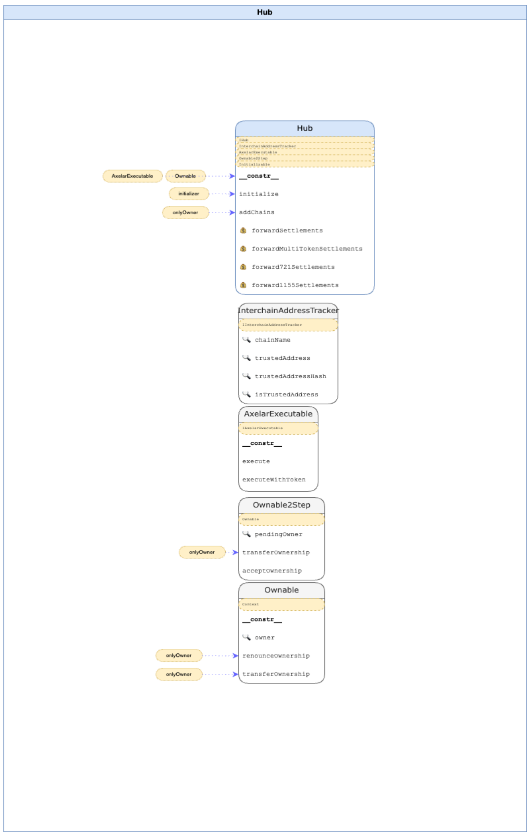
Security Review

Coral

Disclaimer

Birds Eye





Executive Summary

- **Review Period**: 5+1 days review + report
- **Start**: Wed 20 Mar 2024
- **Delivery**: Tue 02 Apr 2024
- **Mitigations Review**: Wed 01 May 2024 - Fri 03 May 2024

Timeline & Scope

- **Initial scope**: https://github.com/0xsquid/n-var-coral-audit-scope
- **Updated scope**: new branch/repo: https://github.com/0xsquid/squid-coral/tree/update
  we shared initial vulnerability information with the team:
  - potential reentrancy / checks effects violations
  - potential fee-on-token problems

- This is the branch with the version of contracts for auditing
  https://github.com/0xsquid/squid-coral/tree/update
- **Updated scope**: new branch/repo:
  https://github.com/0xsquid/squid-coral/tree/develop
  Hey y'all, thanks for the call. You can find the latest at
  https://github.com/0xsquid/squid-coral/tree/develop
  In the live deployment we'll use CREATE3 for deterministic deploy addresses, therefore:
  In 'Hub.sol', lines 52-86 will be removed and replaced with lines 29-31 and 34-47 (setting chainIds, chainNames, and spokeAddresses in the constructor instead of centralized)
  In 'SpokeMulticall.sol', lines 24-26 will be removed and replaced with lines 20-22 (setting spoke address in the constructor instead of centralized)
  That should be the only parts that would change from the current state, in addition to changes as a result of your audit.
- **Engagement Starts**
- **Updated Scope**: new branch/repo:
  https://github.com/0xsquid/squid-coral/tree/feat/offchain-services
  Hi @tkyoooo
  Good day to you! These contracts aren't deployed yet, including the multicall. This multicall is specifically for this implementation, so we're good to make any changes needed wherever needed.
  There's one last update that's been done to the contracts, pushed up recently you can find the latest at
  https://github.com/0xsquid/squid-coral/tree/feat/offchain-services
  These contracts will be deployed via CREATE2, so all constructors have been removed in place of initializers using OpenZeppelin Initializable.
  All of the admin functions have been removed from Hub.sol except for an addChains which is used to just add chain names and ID's according to Axelar's chain names, they'll always use the same Spoke address.
  Any exposed setter functions were removed, those were meant to be in the initializer
- **Mitigations Review**: Updated status of findings, mapping clients responses and changesets including quick-review on whether the changeset sufficiently addresses the findings. **Note:** Please find more information on the resolution for `partially addressed with remarks` tagged findings within the respective findings remediation info box alongside the findings description.

Files in Scope:

- `Hub.sol`
- `Spoke.sol`

Findings

Severity Critical

[CRITICAL][ADDRESSED WITH REMARKS] Spoke - `batchFillOrder` allows filler to provide less `ETH` than required to fill `ERC20_OR_NATIVE` orders

> **[Update] Remediation Note:** Addressed with https://github.com/0xsquid/squid-coral/pull/18/commits/f209a2042bb7e4db1a6d8157ae8b64c2204cc979 by validating enough `ETH` was provided to the call. The client provided the following statement:
>
> > Loops through orders and sums native token amount required for all orders before filling individually.
>
> Note: the fix is very hacky. It does not address the fact that in the `batchFillOrder` case, the `msg.value` check in `fillOrder` is not checking anything. Semantically, this is wrong in `fillOrder` and this technical debt will likely bubble up in a future security review. However, the fact that `msg.value` must be greater than all fill values combined ensures that enough `ETH` was provided a no stored `ETH` is used. It should be noted that it would be much cleaner to refactor the fill logic having a `batchFill` subtracts individual fill amounts from `msg.value` (temporary variable) and revert if it underflows and refactor the `fillOrder` logic into a public function that checks `msg.value` and an internal that executes the fill relying on `msg.value` to be checked already.
>
> Note: the complexity of the fill logic is pretty high with 4 levels of nested if-branches and a try-catch.
>
> ---
>
> `batchFillOrder payable` is a convenience function that takes multiple orders and fills them at once. It calls `fillOrder payable` in a loop to fill the individual orders.
>
> **contracts/Spoke.sol (Lines 230-234):**

```
function batchFillOrder(Order[] calldata orders, Call[][] calldata calls)
external payable {
    for (uint256 i = 0; i < orders.length; i++) {
        fillOrder(orders[i], calls[i]);
    }
}
```

When the asset type is `AssetType.ERC20_OR_NATIVE` it checks that the caller provided enough `ETH` to fill the orders. The problem is, that `msg.value` is static for the duration of the call and does not change when forwarding `ETH` to fill the order. This means, that the filler actually only has to provide enough to match `msg.value < order.fillAmount` once for the highest `order.fillAmount` in the batch. The rest will be paid out from the contracts.balance.

contracts/Spoke.sol (Lines 244-247):

```
if (order.toAssetType == AssetType.ERC20_OR_NATIVE) {
    if (order.toToken == nativeToken) {
        if (msg.value < order.fillAmount) revert InvalidNativeAmount();
```

contracts/Spoke.sol (Lines 264-266):

```
} else {
    payable(order.toAddress).sendValue(order.fillAmount);
}
```

This boils down to a typical iteration using `msg.value` in a loop type of vulnerability that needs to be addressed by implementing a separate accounting to ensure that the filler can only spend what they provided with the `msg.value` for the call.

It should be noted that the general complexity of this function is way too high and a rewrite is suggested. This may include removing the native token support in favor of `ERC20:WETH` transfers.

[CRITICAL][ADDRESSED WITH REMARKS] SpokeMulticall - `_setCallDataParameter` out of bounds mem write; integer overflow

[Update] Remediation Note: Addressed with https://github.com/0xsquid/squid-coral/pull/18/commits/70849a5db8ebee7533f0f84fc58b1d1a7a43fb8a by checking the parameter position. The client provided the following statement:

Checks if parameter position is out of bounds.

Note: Keeping this pattern adds technical debt and complexity. The actual problem of the integer overflow is fixed by a manual check that needs to be called before modifying the parameter. It is not enforced inside the function that actually modifies the parameter. It is recommended to refactor this into something more controllable with guaranteed security that does not rely on developers to call a sequence of functions (check, then modify) correctly. More importantly, the OOB check should accommodate for `mstore` saving an entire word at the end. This means the last valid word is `callData.length - 32 bytes`.

```
if (36 + parameterPosition * 32  > callData.length - 32 /* because mstore
writes a word=32bytes */) return false;
```

The pattern used with `_setCallDataParameter` is really dangerous. The function takes a `memory` reference type and low-level assembly modifies the callers memory data, without performing any checks on it.

- no bounds checks that `(parameterPosition *32 + 36) < callData.length` - this allows out-of-bounds memory to write to any other memory struct in the calls context
- no integer overflow protection - allows to provide a sufficiently large enough `parameterPosition` to wrap the low-level arithmetic operation (integer overflow)

There being no bound check for `parameterPosition` to be within the `callData` means, that, if you fill an order, whoever provides the `calls[]` can write anything anywhere in current mem (including modifying the calls struct itself).

This pattern does not only look bad (backdoor), it is also hard to maintain a codebase with such an unclear side-effect rich pattern and it is clear to say that this will pop up in every future code review because it is a vulnerability pattern. If not already, it will eventually lead to security problems, increase the attack surface for malicious activity, or in the best case just hide errors.

contracts/SpokeMulticall.sol (Lines 30-36):

```
(address token, uint256 amountParameterPosition) = abi.decode(
    call.payload,
    (address, uint256)
);
uint256 amount = IERC20(token).balanceOf(address(this));
if (amount < 2) revert NoTokenAvailable(token);
_setCallDataParameter(call.callData, amountParameterPosition, amount - 1);
```

contracts/SpokeMulticall.sol (Lines 48-56):

```
function _setCallDataParameter(
    bytes memory callData,
    uint256 parameterPosition,
    uint256 value
) private pure {
    assembly {
        mstore(add(callData, add(36, mul(parameterPosition, 32))), value)
    }
}
```

Here's a very simple PoC that illustrate that this pattern can be used to overwrite data from the unrelated `MyMemoryType` from within `_setCallDataParameter` even though the passed memory location is `callData`.

```
contract Storage {
    event logbytes(uint index, bytes a);
    event Log( uint256 left);

    struct MyMemoryType {
            uint256 value;
    }

    function _setCallDataParameter(
        bytes memory callData,
        uint256 parameterPosition,
        uint256 value
    ) private pure  {
        assembly {
            mstore(add(callData, add(36, mul(parameterPosition, 32))), value)
        }
    }

    function doSomethingWithCalldata(bytes memory callData) public  returns
(uint256){
        MyMemoryType memory mem; // dummy struct on mem, we'll overflow into
ths
        mem.value = 1000; // initial value

        emit logbytes(1, callData);
        _setCallDataParameter(callData, 4, 0xce);
        emit logbytes(4, callData);

        return mem.value;
    }

    function callSomething(address target) public returns(bool) {
        (bool success, bytes memory data) = target.call("1111"); //@audit
contract exists
        emit logbytes(1, data);
        require(success);
        return success;
    }
}
```

[CRITICAL][🅾 ADDRESSED WITH REMARKS] Spoke - Lack of Array Length Validation; unchecked `orderHashes`; zero-length inputs

[Update] Remediation Note: Addressed with https://github.com/0xsquid/squid-coral/pull/18/commits/1c1064ea12c4be44e44ffbc0acab543545420ee9 by checking that non-zero orderHashes were provided and recording fees before calling out to the tokens. The client provided the following statement:

Added validation for array lengths/zero lengths.

As for unchecked orderHashes, the order parameters are validated on the Hub contract and sent to the Spoke for efficiency. The Spoke can only _execute with a GMP from the Hub relayed via Axelar; likewise, the Hub can only _execute (process orders) with a GMP from the destination Spoke relayed via Axelar. With this, processed orderHashes derive from paid orders on the destination Spoke. The release is dependent on the orderHash being set to CREATED in the source Spoke. So the tokens within the Order have to be locked on the source Spoke in order to be released. Verification happens cross-chain and is dependent on a relationship between the Hub and Spoke contracts for this system to work, so this pattern is required.

Note: Even though parameters are validated on the connected chain they should be validated on the target, too. The system is set up to trust a set of contracts with the `Hub` being the authority. If a malicious actor can send messages from `Hub` to any of the `Spokes` that don't get validated on the `Spoke` this may be problematic. Consider using a defense in-depth approach that builds multiple layers of defense checking the validity of data on both ends to reduce the attack surface and contain security events.

Multiple functions receive arrays as input parameters but no validation is performed to ensure that these arrays **have the same length** and **are of non-zero length**.

In `_releaseMultiTokenBatched`, `_release721Batched`, and `_release1155Batched` methods, the code loops through the `orderHashes` array, `fromTokens` array, `tokenIds` array, and `fromAmounts` array without checking if all these arrays contain the same number of elements. This could lead to unexpected behavior if arrays of different lengths are passed.

Even more problematic, `orderHashes` and not verified at all and taken as ground truth. Release information is not bound to the hash and hence if the caller allows (cross-chain `Hub`) it may provide `orderHashes` of any number that may be completely unrelated to the tokens released! This is problematic as the trust of this contract (and any updates to internal structures and state) should never rely on the security of another contract.

Note that providing zero-length inputs may allow the caller to skip parts of the function which is highly problematic.

Note: checks effects violation! Update internal accounting first, external call last (`tokenToCollectedFees`).

Example

- `Spoke`

For example, here, `orderHashes` can be zero, no order hash would be updated, but the contract would still pay `filler` `fromAmount`. (dangerous!)

contracts/Spoke.sol (Lines 374-395):

```
function _releaseBatched(
    bytes32[] memory orderHashes,
    address filler,
    uint256 fromAmount,
    uint256 processedFees,
    address token
) internal {
    for (uint256 i = 0; i < orderHashes.length; i++) {
        bytes32 orderHash = orderHashes[i];
        if (orderHashToStatus[orderHash] != OrderStatus.CREATED) revert
EmptyOrder();
        orderHashToStatus[orderHash] = OrderStatus.SETTLED;

        emit TokensReleased(orderHash);
    }

    if (token == nativeToken) {
        payable(filler).sendValue(fromAmount);
    } else {
        IERC20(token).safeTransfer(filler, fromAmount);
    }
    tokenToCollectedFees[token] += processedFees;
}
```

contracts/Spoke.sol (Lines 397-413):

```
function _releaseMultiTokenBatched(
    bytes32[] memory orderHashes,
    address filler,
    uint256[] memory fromAmounts,
    uint256[] memory processedFees,
    address[] memory fromTokens
) internal {
    for (uint256 i = 0; i < orderHashes.length; i++) {
        bytes32 orderHash = orderHashes[i];
        if (orderHashToStatus[orderHash] != OrderStatus.CREATED) revert 
EmptyOrder();
        orderHashToStatus[orderHash] = OrderStatus.SETTLED;

        emit TokensReleased(orderHash);
    }

    for (uint256 i = 0; i < fromTokens.length; i++) {
        if (fromTokens[i] == nativeToken) {
```

contracts/Spoke.sol (Lines 422-437):

```
function _release721Batched(
    bytes32[] memory orderHashes,
    address filler,
    uint256[] memory tokenIds,
    address collection
) internal {
    for (uint256 i = 0; i < orderHashes.length; i++) {
        bytes32 orderHash = orderHashes[i];
        if (orderHashToStatus[orderHash] != OrderStatus.CREATED) revert 
EmptyOrder();
        orderHashToStatus[orderHash] = OrderStatus.SETTLED;

        IERC721(collection).transferFrom(address(this), filler, tokenIds[i]);

        emit TokensReleased(orderHash);
    }
}
```

contracts/Spoke.sol (Lines 439-461):

```
function _release1155Batched(
    bytes32[] memory orderHashes,
    address filler,
    uint256[] memory tokenIds,
    uint256[] memory fromAmounts,
    address collection
) internal {
    for (uint256 i = 0; i < orderHashes.length; i++) {
        bytes32 orderHash = orderHashes[i];
        if (orderHashToStatus[orderHash] != OrderStatus.CREATED) revert 
EmptyOrder();
        orderHashToStatus[orderHash] = OrderStatus.SETTLED;

        IERC1155(collection).safeTransferFrom(
            address(this),
            filler,
            tokenIds[i],
            fromAmounts[i],
            ""
        );

        emit TokensReleased(orderHash);
    }
}
```

Recommendation

- Ensure that input arrays are of non-zero length!
- Ensure that the lengths of all the input arrays are identical before processing them. This can be achieved by adding a condition at the start of each function to compare the lengths of the arrays. If they don't match, revert the transaction. Here is an example:

```
require(orderHashes.length == fromAmounts.length && orderHashes.length ==
fromTokens.length, "Input lengths must be equal");
```

- provide the original orders with the call, calculate the order hash and total token amounts, update the status, then release the tokens only if they belong to a known order hash.

[CRITICAL][✔ FIXED] Reentrancy / Checks-Effects Violation

[Update] Remediation Note: Addressed with https://github.com/0xsquid/squid-coral/pull/18/commits/0fa06914fc36a4690ea0cb8a219db1e6955c0b4d by adhering to the checks-effects pattern and adding contract-wide reentrancy guards to `Hub` (as a safety) and `Spoke` (required). The client provided the following statement:

> Added nonReentrant to all functions handling tokens, moved relevant status updates to before transfers.

Note: `nonReentrant` modifiers are typically set on public/external interfaces. However, it is also totally valid to use them with private/internal functions.

As communicated in a call with the team before, many of the functions in the contract system are not adhering to the Checks-Effects-Interactions Pattern, where external calls are only performed after state-changing operations are made (internal accounting, locks). Furthermore, relevant functions that interact with external contracts are not protected with a contract-wide reentrancy guard. This leads to problems where an untrusted external call may reenter a function to drain contract balance or modify the state in an unintended way.

Note that the signature validation library may hand over control to the caller, too, however, the libraries used here only call `view` only functions resulting in non state-changing staticcalls. However, it is important to be aware of that fact and nevertheless, protect functions adequately.

Examples

- `createOrder` - sets `orderHashToStatus` after ext calls. caller might reenter with the same order hash.

contracts/Spoke.sol (Lines 99-131):

```
function _createOrder(Order calldata order, address fromAddress) private {
    bytes32 orderHash = keccak256(abi.encode(order));

    if (orderHashToStatus[orderHash] != OrderStatus.EMPTY) revert
OrderAlreadyExists();
    if (block.timestamp > order.expiry) revert OrderExpired();
    if (keccak256(bytes(order.fromChain)) !=
keccak256(bytes(_uintToString(_getChainId()))))
        revert InvalidChain();

    if (order.fromAssetType == AssetType.ERC20_OR_NATIVE) {
        if (order.fromAmount == 0) revert InvalidAmount();

        if (order.fromToken == nativeToken) {
            if (msg.value != order.fromAmount) revert InvalidNativeAmount();
        } else {
            IERC20(order.fromToken).safeTransferFrom(fromAddress,
address(this), order.fromAmount);
        }
    } else if (order.fromAssetType == AssetType.ERC721) {
        IERC721(order.fromToken).safeTransferFrom(fromAddress, address(this),
order.tokenId);
    } else if (order.fromAssetType == AssetType.ERC1155) {
        if (order.fromAmount == 0) revert InvalidAmount();
        IERC1155(order.fromToken).safeTransferFrom(
            fromAddress,
            address(this),
            order.tokenId,
            order.fromAmount,
            ""
        );
    }

    orderHashToStatus[orderHash] = OrderStatus.CREATED;

    emit OrderCreated(orderHash, order);
}
```

- `sponsorOrder` - might staticcall signer (if contract) (generally safe) but also performs a token transfer with a potential callback before the state change
- `sponsorOrderUsingPermit2` - performs a token transfer with a potential callback before the state change. sets `orderHashToStatus` after token transfer.

contracts/Spoke.sol (Lines 153-162):

```
permit2.permitWitnessTransferFrom(
    permit,
    transferDetails,
    order.fromAddress,
    witness,
    ORDER_WITNESS_TYPE_STRING,
    signature
);

orderHashToStatus[orderHash] = OrderStatus.CREATED;
```

- `refundOrder` - is doing it right by updating right after checking the order hash.
- `collectFees` - is doing it right by resetting accrued fees before calling the recipient.
- `fillOrder` - sets settlement status after external call. this may be used to steal tokens by filling the same order multiple times.

contracts/Spoke.sol (Lines 226-301):

```
function fillOrder(Order calldata order, Call[] calldata calls) public
payable {
    bytes32 orderHash = keccak256(abi.encode(order));

    if (settlementToStatus[orderHash] != SettlementStatus.EMPTY) revert
OrderAlreadySettled();
    if (msg.sender != order.filler) revert OnlyFillerCanSettle();
    if (keccak256(bytes(order.toChain)) !=
keccak256(bytes(_uintToString(_getChainId()))))
        revert InvalidDestinationChain();

    if (order.toAssetType == AssetType.ERC20_OR_NATIVE) {
        if (order.toToken == nativeToken) {
            if (msg.value < order.fillAmount) revert InvalidNativeAmount();

            if (order.postHookHash != bytes32(0)) {
                bytes memory callsData = abi.encode(calls);
                if (keccak256(callsData) != order.postHookHash) revert
InvalidPostHookProvided();
                payable(address(spokeMulticall)).sendValue(order.fillAmount);

                uint256 initialContractNativeBalance =
address(spokeMulticall).balance - order.fillAmount;

                bool success = spokeMulticall.execute(calls);
                uint256 remainingNativeBalance =
address(spokeMulticall).balance - initialContractNativeBalance;

                if (remainingNativeBalance > 0 || !success) {
                    spokeMulticall.recoverTokens(order.toAddress,
order.toToken, remainingNativeBalance);
                }

            } else {
                payable(order.toAddress).sendValue(order.fillAmount);
            }
        } else {
            if (order.postHookHash != bytes32(0)) {
                bytes memory callsData = abi.encode(calls);
                if (keccak256(callsData) != order.postHookHash) revert
InvalidPostHookProvided();
                IERC20(order.toToken).safeTransferFrom(
                    order.filler,
                    address(spokeMulticall),
                    order.fillAmount
                );

                uint256 initialContractTokenBalance =
IERC20(order.toToken).balanceOf(address(spokeMulticall)) - order.fillAmount;

                bool success = spokeMulticall.execute(calls);
                uint256 remainingTokenBalance =
IERC20(order.toToken).balanceOf(address(spokeMulticall)) -
initialContractTokenBalance;

                if (remainingTokenBalance > 0 || !success) {
                    spokeMulticall.recoverTokens(order.toAddress,
order.toToken, remainingTokenBalance);
                }
            } else {
                IERC20(order.toToken).safeTransferFrom(
                    order.filler,
                    order.toAddress,
                    order.fillAmount
                );
            }
        }
    } else if (order.toAssetType == AssetType.ERC721) {
        IERC721(order.toToken).safeTransferFrom(
            order.filler,
            order.toAddress,
            order.fillTokenId
        );
    } else if (order.toAssetType == AssetType.ERC1155) {
        IERC1155(order.toToken).safeTransferFrom(
            order.filler,
            order.toAddress,
            order.fillTokenId,
            order.fillAmount,
            ""
        );
    }

    settlementToStatus[orderHash] = SettlementStatus.FILLED;

    emit OrderSettled(orderHash, order);
}
```

- `_releaseBatched` and `_releaseMultiTokenBatched` - updates `tokenToCollectedFees[token]` after token transfers

contracts/Spoke.sol (Lines 375-380):

```
if (token == nativeToken) {
    payable(filler).sendValue(fromAmount);
} else {
    IERC20(token).safeTransfer(filler, fromAmount);
}
tokenToCollectedFees[token] += processedFees;
```

contracts/Spoke.sol (Lines 398-404):

```
for (uint256 i = 0; i < fromTokens.length; i++) {
    if (fromTokens[i] == nativeToken) {
        payable(filler).sendValue(fromAmounts[i]);
    } else {
        IERC20(fromTokens[i]).safeTransfer(filler, fromAmounts[i]);
    }
    tokenToCollectedFees[fromTokens[i]] += processedFees[i];
```

Recommendation

It is recommended to always adhere to the Checks-Effects-Interactions pattern. Not only for cases where you are unsure but, in general, for all cases. Adjust internal locks first, and perform external calls to untrusted parties last to avoid that they can exploit stale states. Furthermore, the contract system allows interaction with a variety of tokens that may perform callbacks to a token recipient or sender (`ERC777` sender and receiver callbacks, `ERC20` variants with callbacks, `ERC721 receiver.onERC721Received`, `ERC1155 receiver.onERC1155Received`, …). All these interactions must be seen as untrusted and adequate protection be implemented. It is, therefore, recommended to always guard functions that interact with untrusted parties with a contract-wide reentrancy guard (see open-zeppelin `nonReentrant` modifier).

Severity High

[HIGH][✔ ACKNOWLEDGED] Fee on transfer token support might be problematic

[Update] Remediation Note: The client acknowledged this finding, providing the following statement:

> Our current solution is a stance of non-support. Our backend is responsible for serving up quotes to users for tokens that are supported through this system. It's likely we will simply not support fee on transfer tokens from on our side. Orders can still be crafted and executed with fee on transfer, but it would require manual work and is unlikely for the average user, additionally they would need a filler for the fee on transfer token, which we also don't support.
>
> If we will support fee on transfer tokens in the future, the transfer fee will likely be baked into the quote to reduce the amount the user receives on the destination chain. If refunds are required for fee on transfer tokens that we support, manual

transfers will be needed to ensure the proper amount exists in the source Spoke to perform a refund, this pattern is fine for us. For now, these tokens will not be supported by our system, no on-chain guardrails should be required.

When creating an order the `Spoke` contract escrows `ERC20` tokens. There are a wide variety of different tokens as can be seen in this Token Interaction Checklist from 2020. Specifically the case where the target token is a "fee-on-transfer" token can be problematic, as the actual amount of tokens escrowed will differ from the `order.fromAmount`. If a fee on transfer is taken, the actual amount escrowed will be less than the order amount. Since orders cannot be filled partially, tokens might get stuck.

For example, `msg.sender` will be unable to call `refundOrder` because the contract holds less than `order.fromAmount` tokens, hence, tokens will be stuck. Unless they provide "extra" token (pot. front-run race if they don't do this in the same transaction) they will not be able to get them out.

The same is true for all other methods that are bound to amounts in the order where expectation is that the order amount resides in the contract for methods to be callable.

contracts/Spoke.sol (Lines 122-124):

```
} else {
    IERC20(order.fromToken).safeTransferFrom(fromAddress, address(this), 
order.fromAmount);
}
```

Now, it is up the the participants to understand these limitations. From the filler side, liquidation providers would not fill orders that are not favorable for them. From the users side you would want to be able to get back the tokens you escrowed. Ideally, the contract would prevent users from shooting themselves in the foot by only allowing tokens that are safe to use. This can have an impact on the users' experience. Hence, a variant that can be acceptable is to modify the caller's order to track the actual token amount sent to the contract. This will likely require deeper changes to an already complex system of contracts. Another option may be to allow partial fills. This, however, will introduce a lot more complexity again. Gatekeeping for supported tokens might be the easiest solution. Anyway, it is recommended to familiarize yourself with the Token Interaction Checklist and check how bridges typically prevent fee-on-transfer problems.

[HIGH][🅾 ADDRESSED WITH REMARKS] Hub - Lack of array validation in `forward*Settlements`, `_execute`

[Update] Remediation Note: Addressed with https://github.com/0xsquid/squid-coral/pull/18/commits/ac78f812529888605b6afc2760742198f372a1f2 by enforcing a minimum transaction value and checking for non-zero array length. The client provided the following statement:

> Checks for valid array lengths and requires gas for forwarding. (meaningful amount to be decided)
>
> fromTokens.length shouldn't match orders.length as we may be processing multiple orders with the same token or in many cases only using a single token for all orders. Arrays are used for all cases to standardize decoding on source Spoke.

Note: It is recommended to enforce strong input validation for all methods especially for `forwardSettlement <-> execute` communication. i.e. to only name on, if only one `fromToken` is expected, both methods should enforce only one is present (e.g. `FillType.ERC721`, `FillType.ERC1155`, `FillType.ERC20_OR_NATIVE`) or else extra data will be silently ignored.

All `forward*Settlements` functions fail to validate inputs. For example, the function can be called with an empty `orders` array which will lead to the whole for-loop to be skipped, basically performing no operation on the hub. Instead of aborting in this case, the function continues to send unusable cross-chain messages with empty `orderHashes, fromAmounts, fees, tokenIds`. As this does not make sense, it should be rejected immediately.

- Additionally, there is no array length validation for `fromTokens.length == orders.length`.
- Additionally, there is no enforcement for `msg.value` to provide a meaningful amount of gas for the cross-chain transaction.

contracts/Hub.sol (Lines 70-123):

```
function forwardSettlements(
    Order[] calldata orders,
    string calldata fromChain,
    address[] calldata fromTokens,
    address filler,
    AssetType fromAssetType
) external payable {
    bytes32[] memory orderHashes = new bytes32[](orders.length);
    uint256[] memory fromAmounts = new uint256[](1);
    uint256[] memory fees = new uint256[](1);
    uint256[] memory tokenIds = new uint256[](0);

    for (uint256 i = 0; i < orders.length; i++) {
        bytes32 orderHash = keccak256(abi.encode(orders[i]));
        orderHashes[i] = orderHash;

        if (orderHashToStatus[orderHash] != OrderStatus.VERIFIED) revert
OrderNotSettled();
        if (keccak256(bytes(orders[i].fromChain)) !=
keccak256(bytes(fromChain)))
            revert InvalidSettlementChain();
        if (orders[i].fromAssetType != fromAssetType) revert
InvalidSettlementSourceType();
        if (orders[i].filler != filler) revert InvalidSettlementFiller();
        if (orders[i].fromToken != fromTokens[0]) revert
InvalidSettlementSourceToken();

        uint256 fee = (orders[i].fromAmount * orders[i].feeRate) / 100000;

        orderHashToStatus[orderHash] = OrderStatus.FILLED;
        fees[0] += fee;
        fromAmounts[0] += orders[i].fromAmount - fee;

        emit SettlementFilled(orderHash, orders[i]);
    }

    bytes memory payload = abi.encode(
        FillType.ERC20_OR_NATIVE,
        orderHashes,
        filler,
        tokenIds,
        fromAmounts,
        fees,
        fromTokens
    );
    gasService.payNativeGasForContractCall{value: msg.value}(
        address(this),
        chainIdToChainName[fromChain],
        spoke,
        payload,
        msg.sender
    );
    gateway.callContract(
        chainIdToChainName[fromChain],
        spoke,
        payload
    );
}
```

Also see related findings for the lack of array validation in `Spoke`'s `release*` functions.

The same goes for `_execute` which should bail on running `_processSettlements` on empty `orderHashes`.

contracts/Hub.sol (Lines 299-315):

```
function _execute(
    string calldata fromChain,
    string calldata fromContractAddress,
    bytes calldata payload
) internal virtual override onlyTrustedAddress(fromChain,
fromContractAddress) {
    bytes32[] memory orderHashes = abi.decode(payload, (bytes32[]));
    _processSettlements(orderHashes);
}

function _processSettlements(bytes32[] memory orderHashes) internal {
    for (uint256 i = 0; i < orderHashes.length; i++) {
        if (orderHashToStatus[orderHashes[i]] != OrderStatus.EMPTY) revert
OrderAlreadyProcessed();
        orderHashToStatus[orderHashes[i]] = OrderStatus.VERIFIED;

        emit SettlementProcessed(orderHashes[i]);
    }
}
```

[HIGH][✔ FIXED] Spoke - cannot receive `ERC1155` tokens due to missing implementation of `ERC1155Holder`

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/e6e20fcc7b13b84cb370546e089940c93e5176a7 by inheriting `ERC1155Holder`. The client provided the following statement:

Inherits and uses ERC1155Holder.

Note: This should be covered by unittests.

`createOrder` with `AssetType.ERC1155` will ultimately revert in `ERC1155.safeTransferFrom` because `ERC1155`, if the target is a contract, checks if the contract accepts `ERC1155` tokens by requiring target to return a magic like `target.onERC1155Received() == MAGIC`.

contracts/Spoke.sol (Lines 127-136):

```
} else if (order.fromAssetType == AssetType.ERC1155) {
    if (order.fromAmount == 0) revert InvalidAmount();
    IERC1155(order.fromToken).safeTransferFrom(
        fromAddress,
        address(this),
        order.tokenId,
        order.fromAmount,
        ""
    );
}
```

The contract should implement `ERC1155Holder`, see:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC1155/utils/ERC1155Utils.sol

Severity Medium

[MEDIUM][✔ FIXED] Hub - `forward*Settlements` unchecked `chainIdToChainName[fromChain]`

[Update] Remediation Note: Addressed with https://github.com/0xsquid/squid-coral/pull/18/commits/33e9270404ad4fb1a3f86147df5a360befd74cfb by explicitly checking that `chainName` is a known chain. The client provided the following statement:

Checks for valid fromChain on the Hub.

There is no check to validate that `fromChain` is a known chain. The mapping `chainIdToChainName[fromChain]` returns empty string `` for unknown `chainId`'s which should be a reason to revert instead of proceeding with empty string.

Especially in combination with the finding outlined in this report regarding providing empty `orderHashes` which would even skip all the checks.

contracts/Hub.sol (Lines 20-20):

```
mapping(string => string) public chainIdToChainName;
```

contracts/Hub.sol (Lines 113-113):

```
chainIdToChainName[fromChain],
```

contracts/Hub.sol (Lines 119-119):

```
chainIdToChainName[fromChain],
```

contracts/Hub.sol (Lines 170-170):

```
chainIdToChainName[fromChain],
```

contracts/Hub.sol (Lines 176-176):

```
chainIdToChainName[fromChain],
```

contracts/Hub.sol (Lines 234-234):

`chainIdToChainName[fromChain],`

contracts/Hub.sol (Lines 240-240):

`chainIdToChainName[fromChain],`

contracts/Hub.sol (Lines 287-287):

`chainIdToChainName[fromChain],`

contracts/Hub.sol (Lines 293-293):

`chainIdToChainName[fromChain],`

[MEDIUM][🅾 ADDRESSED WITH REMARKS] SpokeMulticall - Unclear pattern

[Update] Remediation Note: Addressed with https://github.com/0xsquid/squid-coral/pull/18/commits/fbad55aad6284a6da0c3bcebaf9bae94b7fd23dd by removing the `send/receive()` logic, passing the value with the function call directly. The client provided the following statement:

> Changed fillOrder multicall execute pattern for native tokens to send the value along with the execute call.

Note: As noted with this initial finding: "this looks way too complicated for what it tries to achieve [...]". Complexity of for this function is way to high to be operated securely. It is recommended to significantly reduce complexity and simplify the logic.

Note: tryCatch can be forced to take the catch path for OutOfGas scenarios (1/64 gas left for other operations; typically this should not be enough to cause much harm but theoretically can) or if the caller deliberately forces a revert (OutOfBoundParameter, NoTokenAvailable, CallFailed).

1. We would suggest to check for the specific error you want to catch in the catch clause and revert on unknown errors. conveying business logic with errors in catch clauses can be very problematic.
2. this will send fillAmount to `to` on `outOfBounds` errors. (see tryCatch remark (1)). Due to function complexity it is unclear if this is intended.

It is unclear why `SpokeMulticall` needs a generic `receive()` function and why `Spoke` sends a value transfer first just to then call `execute() payable` without providing a `msg.value` at all.

The whole pattern looks problematic and the intention is unclear.

- why not `execute{value: ...}`
- why calculating `uint256 initialContractNativeBalance = address(spokeMulticall).balance - order.fillAmount` after sending `order.fillAmount` instead of taking the balance before.
- this looks too complicated for what it tries to achieve but it is also unclear what it should achieve.

contracts/Spoke.sol (Lines 251-262):

```
payable(address(spokeMulticall)).sendValue(order.fillAmount);

uint256 initialContractNativeBalance = address(spokeMulticall).balance -
order.fillAmount;

try spokeMulticall.execute(calls) {
    uint256 remainingNativeBalance = address(spokeMulticall).balance -
initialContractNativeBalance;
    if (remainingNativeBalance > 0) {
        spokeMulticall.recoverTokens(order.toAddress, order.toToken,
remainingNativeBalance);
    }
} catch {
    spokeMulticall.recoverTokens(order.toAddress, order.toToken,
address(spokeMulticall).balance - initialContractNativeBalance);
}
```

[MEDIUM][✔ ACKNOWLEDGED] Lack of Documentation / Inline Comments

[Update] Remediation Note: The client acknowledged this finding, providing the following statement:

Natspec descriptions, inline commentation, general documentation is in progress and will be added to all functions of all contracts.

There is a dangerous lack of documentation, natspec function descriptions and inline commentation.

[MEDIUM][✔ FIXED] `_createOrder()` should revert if `order.fromToken != nativeToken` and `msg.value != 0`

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/a3c9274ddcdd30ae742b57ffda73f0290c605956 by reverting if `if (order.fromToken != nativeToken && msg.value != 0)`. The client provided the following statement:

Reverts for unexpected native tokens in orders.

This would prevent users from losing funds by mistake if the payload is incorrect (effectively, they would be donating funds to the contract).

[MEDIUM][✔ FIXED] `Spoke.sponsorOrder()` should not be marked as `payable`

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/a3b5fd376973db7dcc12b78ba6469639de018d0a by removing `payable`. The client provided the following statement:

Removed payable from sponsorOrder as native tokens aren't supported through relaying.

The function `sponsorOrder()` in the `Spoke` contract reverts if `order.fromToken == nativeToken`. Thus, it should not accept a non-zero `msg.value`. One should remove the `payable` modifier.

contracts/Spoke.sol (Lines 93-93):

```
function sponsorOrder(Order calldata order, bytes calldata signature)
external payable {
```

[MEDIUM][✔ FIXED] Use `safeTransfer` (resp. `safeTransferFrom`) instead of `transfer` (resp. `safeTransfer`)

[Update] Remediation Note: Addressed with https://github.com/0xsquid/squid-coral/pull/18/commits/624d1b2676a1c922f0ba8e53cf7cb49fe1bfe9af by using `safeTransfer` instead. The client provided the following statement:

Uses safeTransfer where previously did not.

- In the `Spoke` contract, `transferFrom()` is used to transfer ERC721 in several places. Utilizing the `transferFrom` function might pose a potential security risk because it does not check whether the receiving address can accept ERC721 tokens. The tokens could get locked forever if they are sent to a contract that is not prepared to handle them.
- In `SpokeMulticall`, the `recoverTokens` function utilizes the `transfer` function to transfer tokens to the recipient but does not check the return value of the `transfer` function. Per the ERC20 token standard, the transfer function should return a boolean value indicating the success or failure of the operation. Not checking the return value might lead to unexpected behavior if the transfer function fails; the contract assumes the transfer was successful and continues execution while tokens weren't transferred.

Examples

ERC721:

contracts/Spoke.sol (Lines 433-433):

```
IERC721(collection).transferFrom(address(this), filler, tokenIds[i]);
```

contracts/Spoke.sol (Lines 195-195):

```
IERC721(order.fromToken).transferFrom(address(this), order.toAddress,
order.tokenId);
```

ERC20:

contracts/SpokeMulticall.sol (Lines 66-68):

```
} else {
    IERC20(tokenAddress).transfer(recipient, amount);
}
```

contracts/Spoke.sol (Lines 209-222):

```
function collectFees(address[] calldata tokens) external onlyFeeCollector {
    for (uint256 i = 0; i < tokens.length; i++) {
        uint256 feeAmount = tokenToCollectedFees[tokens[i]];
        if (feeAmount > 0) {
            tokenToCollectedFees[tokens[i]] = 0;
            if (tokens[i] == nativeToken) {
                payable(feeCollector).sendValue(feeAmount);
            } else {
                if (!IERC20(tokens[i]).transfer(feeCollector, feeAmount))
                    revert TokenReleaseFailed();
            }
        }
    }
}
```

Recommendation

- For ERC721 transfers, instead of using `transferFrom`, use the `safeTransferFrom` function. This function performs the same function but will also include an additional check to see if the contract receiving the tokens is capable of handling ERC721 tokens. This provides an additional layer and ensures the safety of the token transfer. If it's a contract, it must implement `onERC721Received` function to receive the tokens.

- For ERC20, either check the value returned by the `transfer` function, or use `safeTransfer` from OpenZeppelin's SafeERC20 library.

Severity Low

[LOW][✔ FIXED] SpokeMulticall - `execute` unused return parameter

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/12d13f7e981b3437ee51f35cf7c604e5a70e7f58 by removing the `bool` return value. The client provided the following statement:

Removed unused return bool from SpokeMulticall execute.

`SpokeMulticall.excute` returns `bool`, however, the returnvalue is hardcoded to `true` as the function reverts on errors, and, the callers never read the return value.

contracts/SpokeMulticall.sol (Lines 24-28):

```
function execute(Call[] memory calls) external payable override onlySpoke
returns (bool) {
    for (uint256 i = 0; i < calls.length; i++) {
        Call memory call = calls[i];
```

contracts/SpokeMulticall.sol (Lines 43-45):

```
}

return true;
```

Here's an example of `Spoke` calling `execute`. It never checks for the `bool` return value.

contracts/Spoke.sol (Lines 255-262):

```
try spokeMulticall.execute(calls) {
    uint256 remainingNativeBalance = address(spokeMulticall).balance -
initialContractNativeBalance;
    if (remainingNativeBalance > 0) {
        spokeMulticall.recoverTokens(order.toAddress, order.toToken,
remainingNativeBalance);
    }
} catch {
    spokeMulticall.recoverTokens(order.toAddress, order.toToken,
address(spokeMulticall).balance - initialContractNativeBalance);
}
```

It is recommended, to remove the `bool` return value from the `SpokeMulticall.execute` function signature as the function reverts on error anyway.

[LOW][✔ ACKNOWLEDGED] Spoke - `forwardSettlements` check for `msg.value` can be bypassed by sending 1 wei

[Update] Remediation Note: The client acknowledged this finding, providing the following statement:

Sufficient gas minimums for `Hub` and `Spoke forward*` Axelar GMP's are still being decided.

The function should probably enforce meaningful minimums as gas stipends for the cross chain call as the `msg.value != 0` check can easily be bypassed.

contracts/Spoke.sol (Lines 317-337):

```
function forwardSettlements(bytes32[] calldata orderHashes) external payable
{
    if (msg.value == 0) revert GasRequired();

    for (uint256 i = 0; i < orderHashes.length; i++) {
        if (settlementToStatus[orderHashes[i]] != SettlementStatus.FILLED)
            revert OrderNotSettled();
        settlementToStatus[orderHashes[i]] = SettlementStatus.FORWARDED;

        emit SettlementForwarded(orderHashes[i]);
    }

    bytes memory payload = abi.encode(orderHashes);
    gasService.payNativeGasForContractCall{value: msg.value}(
        address(this),
        hubChainName,
        hubAddress,
        payload,
        msg.sender
    );
    gateway.callContract(hubChainName, hubAddress, payload);
}
```

[LOW][🅾 ADDRESSED WITH REMARKS] Hub - `addChains` but no way to remove chains

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/ce7bbb997fc3f88b5335a7918211b41bb9de2212 by adding means to remove chains from the hub. The client provided the following statement:

> Added removeChains admin function to Hub.

Note: `onlyOwner` can frontrun transactions deliberately blocking them by removing the chain. Ensure `onlyOwner` is a delayed multisig or DAO with an announcement.

Note: `removeChains` should emit an event.

There is a function to `addChains` but no way to remove previously added chains.

contracts/Hub.sol (Lines 53-62):

```
function addChains(
    string[] calldata chainIds,
    string[] calldata chainNames
) external onlyOwner {
    if (chainIds.length != chainNames.length) revert InvalidInputData();
    for (uint256 i = 0; i < chainIds.length; i++) {
        chainIdToChainName[chainIds[i]] = chainNames[i];
        _setTrustedAddress(chainNames[i], spoke);
    }
}
```

[LOW][🅾 ADDRESSED WITH REMARKS] Emit events for state changing operations

[Update] Remediation Note: Addressed by adding more events with https://github.com/0xsquid/squid-coral/pull/18/commits/1e3fd3af16eed5116dfc299e6e40c5d204746077.

Note: Some functions may emit events even though nothing changed (e.g. `addChains` with empty array)

State-changing functions like `initialize`, `addChains` should emit a distinct event (audit trail).

- `Hub`

contracts/Hub.sol (Lines 52-62):

```
function addChains(
    string[] calldata chainIds,
    string[] calldata chainNames
) external onlyOwner {
    if (chainIds.length != chainNames.length) revert InvalidInputData();
    for (uint256 i = 0; i < chainIds.length; i++) {
        chainIdToChainName[chainIds[i]] = chainNames[i];
        _setTrustedAddress(chainNames[i], spoke);
    }
}
```

contracts/Hub.sol (Lines 35-45):

```
function initialize(
    address _gateway,
    address _gasService,
    address _owner,
    string memory _spoke
) external initializer {
    transferOwnership(_owner);
    gateway = IAxelarGateway(_gateway);
    gasService = IAxelarGasService(_gasService);
    spoke = _spoke;
}
```

- `SpokeMulticall` - does not emit explicit events at all

contracts/SpokeMulticall.sol (Lines 21-24):

```
function initialize(address _spoke) external initializer {
    spoke = _spoke;
}
```

- `Spoke` - `initialize`, `collectFees`

[LOW][✔ FIXED] Spoke - asset action selection should revert by default (instead of relying on enum conversion to fail)

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/546e79b69bd52e8d849be2ea450de25ee729174c by explicitly reverting on the else branch. The client provided the following statement:

Revert for invalid asset types in the else branch of functions that use asset selection.

Asset action selection should logically `revert()` in else branch, just to ensure, that even in the future with pot changes to the code, no one can create orders with illegal enum values. Note that right now the implicit enum conversion will revert on invalid enum values, but it is much cleaner to have explicit else branches for unmatched assets that always clearly revert, even if it's only for code documentation.

contracts/Spoke.sol (Lines 117-136):

```
if (order.fromAssetType == AssetType.ERC20_OR_NATIVE) {
    if (order.fromAmount == 0) revert InvalidAmount();

    if (order.fromToken == nativeToken) {
        if (msg.value != order.fromAmount) revert InvalidNativeAmount();
    } else {
        IERC20(order.fromToken).safeTransferFrom(fromAddress, address(this),
order.fromAmount);
    }
} else if (order.fromAssetType == AssetType.ERC721) {
    IERC721(order.fromToken).safeTransferFrom(fromAddress, address(this),
order.tokenId);
} else if (order.fromAssetType == AssetType.ERC1155) {
    if (order.fromAmount == 0) revert InvalidAmount();
    IERC1155(order.fromToken).safeTransferFrom(
        fromAddress,
        address(this),
        order.tokenId,
        order.fromAmount,
        ""
    );
}
```

contracts/Spoke.sol (Lines 187-204):

```
if (order.fromAssetType == AssetType.ERC20_OR_NATIVE) {
    if (order.fromToken == nativeToken) {
        payable(order.toAddress).sendValue(order.fromAmount);
    } else {
        IERC20(order.fromToken).safeTransfer(order.toAddress,
order.fromAmount);
    }
} else if (order.fromAssetType == AssetType.ERC721) {
    IERC721(order.fromToken).transferFrom(address(this), order.toAddress,
order.tokenId);
} else if (order.fromAssetType == AssetType.ERC1155) {
    IERC1155(order.fromToken).safeTransferFrom(
        address(this),
        order.toAddress,
        order.tokenId,
        order.fromAmount,
        ""
    );
}
```

[LOW][✔ FIXED] Hub / Spoke - misleading revert message

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/1c678610e9939e92bef28ab777c971e11f4a9970 by renaming the error. The client provided the following statement:

Clear revert messages for onlyTrustedAddress.

Either the modifier `onlyTrustedAddress` or the revert message `error onlyHub` should be changed to fit the modifiers/errors name. If there is only one trusted address, consider changing the error to `onlyTrustedAddress`

contracts/Spoke.sol (Lines 51-54):

```
modifier onlyTrustedAddress(string calldata fromChainName, string calldata
fromContractAddress) {
    if (!isTrustedAddress(fromChainName, fromContractAddress)) revert
OnlyHub();
    _;
}
```

contracts/Hub.sol (Lines 22-25):

```
modifier onlyTrustedAddress(string calldata fromChainName, string calldata
fromContractAddress) {
    if (!isTrustedAddress(fromChainName, fromContractAddress)) revert
OnlySpoke();
    _;
}
```

[LOW][🅾 ADDRESSED WITH REMARKS] Use best type available in declaration (also function args) instead of casting from addr

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/d54a387744c317497f250cd1f0c3be97f7b44bb8. The client provided the following statement:

Uses best type available for initialize arguments.

Couldn't find other instances besides IERC20 for token addresses, however the addresses in these functions are used in other ways before typecasting.

Note: Events can be declared with contract types, too, removing the need to cast them.

Declare state variables with the best type available and downcast to `address` if needed. Typecasting inside the corpus of a function is unneeded when the parameter's type is known beforehand. Declare the best type in function arguments, state vars. Always return the best type available instead of falling back to `address`.

Some examples (more instances in the code):

contracts/Spoke.sol (Lines 64-72):

```
function initialize(
    address _gateway,
    address _gasService,
    address _permit2,
    address payable _spokeMultiCall,
    address _feeCollector,
    string memory _hubChainName,
    string memory _hubAddress
) external initializer {
```

contracts/Hub.sol (Lines 35-45):

```
function initialize(
    address _gateway,
    address _gasService,
    address _owner,
    string memory _spoke
) external initializer {
    transferOwnership(_owner);
    gateway = IAxelarGateway(_gateway);
    gasService = IAxelarGasService(_gasService);
    spoke = _spoke;
}
```

[LOW] Adhere to Solidity Coding Guidelines

Some instances of inconsistent coding style were identified in the code base. Specifically:

- Consider moving the `nativeToken` constant to a shared contract instead of re-declaring it whenever needed.
- In conformance with the solidity code style guidelines, declare `constant` variabls in all uppercase letters

contracts/SpokeMulticall.sol (Lines 13-13):

```
address public constant nativeToken =
0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
```

contracts/Spoke.sol (Lines 32-36):

```
address public constant nativeToken =
0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE;
bytes32 public constant ORDER_TYPEHASH = keccak256("Order(address
fromAddress,address toAddress,address filler,address fromToken,address
toToken,uint256 expiry,uint256 fromAssetType,uint256 toAssetType,uint256
fromAmount,uint256 tokenId,uint256 fillAmount,uint256 fillTokenId,uint256
feeRate,string fromChain,string toChain,bytes32 postHookHash)");
string public constant ORDER_WITNESS_TYPE_STRING =
    "Order witness)Order(address fromAddress,address toAddress,address
filler,address fromToken,address toToken,uint256 expiry,uint256
fromAssetType,uint256 toAssetType,uint256 fromAmount,uint256 tokenId,uint256
fillAmount,uint256 fillTokenId,uint256 feeRate,string fromChain,string
toChain,bytes32 postHookHash)TokenPermissions(address token,uint256 amount)";
```

- Put if-branch bodies into a code-block (`{}`), especially if they are on the next line. This could be easily overlooked by another dev in future iterations.

contracts/Spoke.sol (Lines 114-115):

```
if (keccak256(bytes(order.fromChain)) !=
keccak256(bytes(_uintToString(_getChainId()))))
    revert InvalidChain();
```

contracts/Spoke.sol (Lines 98-104):

```
if (
    !SignatureChecker.isValidSignatureNow(
        order.fromAddress,
        _hashTypedDataV4(_hashOrderTyped(order)),
        signature
    )
) revert InvalidUserSignature();
```

- Source Unit Structure: External functions first, unrelated internal utility functions last in the source unit.
- Instead of checking for `amount < 2` it would be more clear to check for `amount <= 1`

contracts/SpokeMulticall.sol (Lines 35-35):

```
if (amount < 2) revert NoTokenAvailable(token);
```

To favor readability, consider always following a consistent style throughout the code base. We suggest using Solidity's Style Guide as a reference.

[LOW][✔ FIXED] SpokeMulticall - `execute` missing check if contract exists

[Update] Remediation Note: Fixed with https://github.com/0xsquid/squid-coral/pull/18/commits/0cf0acccd1fdade3d534c5b86ac24690c4b1d5fc by always checking that the target is a contract using low level assembly. The client provided the following statement:

Checks if call contract exists in SpokeMulticall.

Note: instead of reimplementing `extcodesize` in assembly one could also use solidity high-level `call.target.codesize > 0`.

`SpokeMulticall` should check that, if `call.calldata.length > 0` then `call.target.codesize > 0` because the intention of providing calldata is that this will be executed in the context of a function invocation on a contract. If the target is no contract but calldata is provided the function call will currently succeed without an error, potentially hiding the fact that this target was not supposed to be called.

contracts/SpokeMulticall.sol (Lines 25-46):

```
function execute(Call[] memory calls) external payable override onlySpoke
returns (bool) {
    for (uint256 i = 0; i < calls.length; i++) {
        Call memory call = calls[i];

        if (call.callType == CallType.FullTokenBalance) {
            (address token, uint256 amountParameterPosition) = abi.decode(
                call.payload,
                (address, uint256)
            );
            uint256 amount = IERC20(token).balanceOf(address(this));
            if (amount < 2) revert NoTokenAvailable(token);
            _setCallDataParameter(call.callData, amountParameterPosition,
amount - 1);
        } else if (call.callType == CallType.FullNativeBalance) {
            call.value = address(this).balance;
        }

        (bool success, bytes memory data) = call.target.call{value:
call.value}(call.callData);
        if (!success) revert CallFailed(i, data);
    }

    return true;
}
```

Severity Info

[INFO] Pin Solidity compiler version

Pin solidity version a specific or range of solidity versions supported. `^0.8.0` risks that the code is accidentally compiled and deployed with a pot. vulnerable compiler version back to `v0.8.0`.

contracts/Spoke.sol (Lines 1-2):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
```

[INFO] [Recommendation] Reentrancy and interaction with untrusted external addresses

Add `nonReentrant` modifiers to all functions that handle Tokens or may call out to untrusted third parties. Additionally, ensure to to adhere to the checks-effects-pattern, always.

[INFO] Create a shared pure function that calculates `orderHash`

Implement a function `getOrderHash(order)` that returns the hash instead of manually calculating it in the various sub-functions. This will improve readability and reduce the chances of introducing errors because a descriptive function is used instead of low-level abi-encode and keccak.

contracts/Spoke.sol (Lines 178-178):

```
bytes32 orderHash = keccak256(abi.encode(order));
```

[INFO] [Note] Any token stuck in `SpokeMulticall` can be swept by anyone by crafting an appropriate order

The `fillOrder()` function within the `Spoke` contract is designed to ensure that in case the order calls fail, only the tokens sent during the current transaction are recovered and sent to the destination by performing balance checks. However, it's important to recognize that this mechanism is not foolproof. Given the function permits the execution of arbitrary calls on the `SpokeMulticall` contract, it's possible for someone to craft a specific order that, for example, invokes the `transfer` function of an ERC20 token, thereby enabling the sweeping of any tokens remaining in the `SpokeMulticall` contract.

contracts/Spoke.sol (Lines 248-262):

```
if (order.postHookHash != bytes32(0)) {
    bytes memory callsData = abi.encode(calls);
    if (keccak256(callsData) != order.postHookHash) revert
InvalidPostHookProvided();
    payable(address(spokeMulticall)).sendValue(order.fillAmount);

    uint256 initialContractNativeBalance = address(spokeMulticall).balance -
order.fillAmount;

    try spokeMulticall.execute(calls) {
        uint256 remainingNativeBalance = address(spokeMulticall).balance -
initialContractNativeBalance;
        if (remainingNativeBalance > 0) {
            spokeMulticall.recoverTokens(order.toAddress, order.toToken,
remainingNativeBalance);
        }
    } catch {
        spokeMulticall.recoverTokens(order.toAddress, order.toToken,
address(spokeMulticall).balance - initialContractNativeBalance);
    }
```

[INFO] Hub / Spoke - inheritance order

Consider keeping the order of inherited contracts similar without mixing it too much between `Hub` and `Spoke`.

contracts/Hub.sol (Lines 13-13):

```
contract Hub is IHub, InterchainAddressTracker, AxelarExecutable,
Ownable2Step, Initializable {
```

contracts/Spoke.sol (Lines 23-25):

```
contract Spoke is ISpoke, AxelarExecutable, ERC721Holder,
InterchainAddressTracker, EIP712, Initializable {
    using Address for address payable;
    using SafeERC20 for IERC20;
```

[INFO] Consider using a generic self-multicall for batch operations

Consider using open-zeppelins `Multicall` library instead of building purpose built batch functions. Note to be careful when using `msg.value` in batch iterations.

contracts/Spoke.sol (Lines 230-235):

```
function batchFillOrder(Order[] calldata orders, Call[][] calldata calls)
external payable {
    for (uint256 i = 0; i < orders.length; i++) {
        fillOrder(orders[i], calls[i]);
    }
}
```

[INFO] Hub / Spoke - Gas efficiency: consider storing chainIds as uints instead of strings

Comparing ASCII representations of uints is not very efficient and it is recommended to completely remove the string dependency for chainIds and fall back to uints instead. This would reduce contract complexity a lot and would get rid of the gas intensive `string->uint->bytes->keccak` conversion.

contracts/Spoke.sol (Lines 114-115):

```
if (keccak256(bytes(order.fromChain)) !=
keccak256(bytes(_uintToString(_getChainId()))))
    revert InvalidChain();
```

remove this:

contracts/Spoke.sol (Lines 491-513):

```
function _uintToString(uint256 value) private pure returns (string memory) {
    if (value == 0) {
        return "0";
    }

    uint256 temp = value;
    uint256 digits;

    while (temp != 0) {
        digits++;
        temp /= 10;
    }

    bytes memory buffer = new bytes(digits);

    while (value != 0) {
        digits -= 1;
        buffer[digits] = bytes1(uint8(48 + (value % 10)));
        value /= 10;
    }

    return string(buffer);
}
```

Hub:

contracts/Hub.sol (Lines 20-20):

```
mapping(string => string) public chainIdToChainName;
```

contracts/Hub.sol (Lines 53-62):

```
function addChains(
    string[] calldata chainIds,
    string[] calldata chainNames
) external onlyOwner {
    if (chainIds.length != chainNames.length) revert InvalidInputData();
    for (uint256 i = 0; i < chainIds.length; i++) {
        chainIdToChainName[chainIds[i]] = chainNames[i];
        _setTrustedAddress(chainNames[i], spoke);
    }
}
```

[INFO] Spoke - consider disallowing dest chain = this.chainId

Consider not allowing the placement of same-chain orders if the purpose of the system is to settle cross-chain orders.

contracts/Spoke.sol (Lines 89-91):

```
function createOrder(Order calldata order) external payable {
    _createOrder(order, msg.sender);
}
```

[INFO] Initialization pattern - ensure to always initialize on deployment (same transaction)

The current initialization pattern relies on an unprotected `initialize()` function, necessitating the deployer to initialize the contract within the deploying transaction. Deploy scripts within the repository appropriately employ the `create2Deployer.deployAndInit()` function, ensuring correct initialization within the same transaction.

It's imperative to acknowledge that any future transition to a proxy pattern should still maintain deploy and initialization within the same transaction to mitigate the risk of frontrunning attacks. Furthermore, for typical proxy deployments, it's essential to ensure that the base implementation is petrified, thereby rejecting any attempts to initialize the base implementation. This precautionary measure is crucial for safeguarding against potential security threats.

[INFO] [Recommendation] General Complexity - consider using WETH instead adding extra complexity for ETH native functionality

The deployed contracts feature significant complexity due to support for native `ETH` and tokens where `ETH` is a special case with the `ERC20` token transfer routines. This complexity increases the likelihood of introducing vulnerabilities. It is recommended to explore alternatives such as Wrapped Ether (WETH) to simplify the contract structure and potentially reduce the attack surface.

[INFO] [Recommendation] - Do not use infinite approvals to `Spoke.sol` in the frontend

The contracts appear to be utilizing the infinite approval pattern, which is a practice we strongly advise against. Utilizing this pattern can expose users to significant risks, especially if vulnerabilities are discovered within the contracts, potentially resulting in the loss of user funds. This approach has previously resulted in severe issues in other bridge systems. To enhance the security of the contracts and protect users, we recommend only granting approval for the specific amount of funds needed for each bridging transaction. This precaution can help prevent potential exploits and mitigate risks associated with the contracts.

Cf. Socket Exploit