Connext Security Review

Auditors

0xLeastwood, Lead Security Researcher

Jonah1005, Lead Security Researcher

Blockdev, Apprentice

Tqts, Apprentice

Gerard Persoon, Quality Officer

Report prepared by: Pablo Misirov

August 30, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Connext is a crosschain liquidity network that enables fast, fully-noncustodial transfers between EVM-compatible chains and L2 systems. It leverages the Ethereum blockchain along with groundbreaking distributed systems tech to enable instant, near-free transfers anywhere in the world.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of CONNEXT according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 15 days in total, Connext engaged with Spearbit to review C4 fixes and Connext NXTP. In this period of time a total of 75 issues were found.

Note that commit hashes were changed during the course of the engagement. While the security team started auditing against the Initial Commit hash, this was quickly switched to the Final Commit hash provided by the Connext team. It must be assumed that all issues have been found against the Final Commit hash.

Find the protocol architecture diagram inside the appendix or navigate there directly by clicking the following number: 6.

Summary

Project Name	Connext
Repository	nxtp
Initial Commit	7f10eca4f6a290...
Final Commit	16ee2f8b441e80...
Type of Project	Crosschain liquidity, Bridge
Audit Timeline	June 27th - July 18th
Engagement 1	C4 fix review
Engagement 2	nxtp audit
Methods	Manual Review

Issues Found

Critical Risk	4
High Risk	16
Medium Risk	10
Low Risk	11
Gas Optimizations	5
Informational	29
Total Issues	75

5 Findings

5.1 Critical Risk

5.1.1 Lack of `transferId` Verification Allows an Attacker to Front-Run Bridge Transfers

Severity: *Critical Risk*

Context: NomadFacet.sol#L99-L149, BridgeRouter.sol#L176-L199, BridgeRouter.sol#L347-L381

Description: The `onReceive()` function does not verify the integrity of `transferId` against all other parameters. Although the `onlyBridgeRouter` modifier checks that the call originates from another BridgeRouter (assuming a correct configuration of the whitelist) to the `onReceive()` function, it does not check that the call originates from another Connext Diamond.

Therefore, allowing anyone to send arbitrary data to `BridgeRouter.sendToHook()`, which is later interpreted as the `transferId` on Connext's `NomadFacet.sol` contract.

This can be abused by a front-running attack as described in the following scenario:

- Alice is a bridge user and makes an honest call to transfer funds over to the destination chain.
- Bob does not make a transfer but instead calls the `sendToHook()` function with the same `_extraData` but passes an `_amount` of `1 wei`.
- Both Alice and Bob have their tokens debited on the source chain and must wait for the Nomad protocol to optimistically verify incoming `TransferToHook` messages.
- Once the messages have been replicated onto the destination chain, Bob processes the message before Alice, causing `onReceive()` to be called on the same `transferId`.
- However, because `_amount` is not verified against the `transferId`, Alice receives significantly less tokens and the `s.reconciledTransfers` mapping marks the transfer as reconciled. Hence, Alice has effectively lost all her tokens during an attempt to bridge them.

```
function onReceive(
  uint32, // _origin, not used
  uint32, // _tokenDomain, not used
  bytes32, // _tokenAddress, of canonical token, not used
  address _localToken,
  uint256 _amount,
  bytes memory _extraData
) external onlyBridgeRouter {
  bytes32 transferId = bytes32(_extraData);
  // Ensure the transaction has not already been handled (i.e. previously reconciled).
  if (s.reconciledTransfers[transferId]) {
    revert NomadFacet__reconcile_alreadyReconciled();
  }

  // Mark the transfer as reconciled.
  s.reconciledTransfers[transferId] = true;
```

Note: the same issues exists with `_localToken`. As a result a malicious user could perform the same attack by using a malicious token contract and transferring the same amount of tokens in the call to `sendToHook()`.

Recommendation: Verify that the call originates from the Connext Diamond on the originator chain. In function `reconcile()` verify that `transferId` is indeed a hash of the other parameters.

Connext: Solved in PR 1630 and PR 1678.

Spearbit: Note: The `BridgeRouter` and the interface to it has changed quite a lot during and after this audit. As it was out of scope for this audit it is also important to conduct a separate review of that particular code, including the interface to Connext.

Connext: An extra audit for `BridgeRouter` is underway.

Spearbit: Acknowledged.

5.1.2 Use of spot dex price when repay portal debt leads to sandwich attacks

Severity: *Critical Risk*

Context: NomadFacet.sol#L286-L290 NomadFacet.sol#L204-L209

Description: When the `NomadFacet` repays the portal debt it has to convert local assets into adopted assets. It first calulates how many assets it needs to swap and then convert the local assets into the adopted assets.

```
  function _reconcileProcessPortal(
    bytes32 _canonicalId,
    uint256 _amount,
    address _local,
    bytes32 _transferId
  ) private returns (uint256) {
    // Calculates the amount to be repaid to the portal in adopted asset
    (uint256 totalRepayAmount, uint256 backUnbackedAmount, uint256 portalFee) =
↪   _calculatePortalRepayment(
      _canonicalId,
      _amount,
      _transferId,
      _local
    );

    ...
    //@audit totalRepayAmount is dependent on the AMM spot price. The swap will not hit the slippage
    (bool swapSuccess, uint256 amountIn, address adopted) =
↪   AssetLogic.swapFromLocalAssetIfNeededForExactOut(
      _canonicalId,
      _local,
      totalRepayAmount,
      _amount
    );

 function _calculatePortalRepayment(
    bytes32 _canonicalId,
    uint256 _localAmount,
    bytes32 _transferId,
    address _local
 )
    internal
    returns (
      uint256,
      uint256,
      uint256
    )
 {
    // @audit A manipulated spot price might be used. availableAmount might be extremely small
    (uint256 availableAmount, address adopted) = AssetLogic.calculateSwapFromLocalAssetIfNeeded(
      _canonicalId,
      _local,
      _localAmount
    );

    // @audit If there aren't enough funds to repay, the protocol absorbs all the slippage
    if (totalRepayAmount > availableAmount) {
    ...
        backUnbackedAmount = availableAmount;
        portalFee = 0;
    ...
```

```
    totalRepayAmount = backUnbackedAmount + portalFee;
  ...
  return (totalRepayAmount, backUnbackedAmount, portalFee);
}
```

The `_calculatePortalRepayment` function calculates the debt to be repaid, `totalRepayAmount`. It also calculates the post-swap amount,`availableAmount`. If the post-swap amount is enough to pay for the outlying debt, `totalRepayAmount` equals the outlying debt. If not, it equals`availableAmount`. Since the `totalRepayAmount` is always smaller than the post-swap amount `availableAmount` which is derived from the AMM price, the swap will not hit the slippage even if the price is off.

Assume the price is manipulated to 1:100. `availableAmount` and `totalRepayAmount` would both approximate to `_amount / 100`. The swap will not hit the slippage as `_amount` can convert to `_amount / 100` in this case.

Exploiters can manipulate the DEX and launch a sandwich attack on every repayment. This can also be done on different chains, making the attackers millions in potential profit.

Connext: We have decided to lean on the policy that Aave portals will not be automatically repaid.

Adding in the automatic repayment of portals adds complexity to the core codebase and leads to issues. Even with the portal repayment in place, issues such as a watcher disconnecting the `xapp` for an extended period mean we have to support out of band repayments regardless. By leaning on this as the only method of repayment, we are able to reduce the code complexity on reconcile. Furthermore, it does not substantially reduce trust. Aave portals essentially amount to an unsecured credit line, usable by bridges. If the automatic repayment fails for any reason (i.e. due to bad pricing in the AMM), then the LP associated with the loan must be responsible for closing out the position in a trusted way.

Solved in PR 1585 by removing this code.

Spearbit: Verified.

5.1.3 `swapOut` allows overwrite of token balance

Severity: *Critical Risk*

Context: StableSwapFacet.sol#L266-L281, SwapUtils.sol#L740-L781, SwapUtils.sol#L417-L473

Description: The `StableSwapFacet` has the function `swapExactOut()` where a user could supply the same `assetIn` address as `assetOut`, which means the `TokenIndex`es for `tokenIndexFrom` and `tokenIndexTo` function `swapOut()` are the same.

In function `swapOut()` a temporay array is used to store `balances`. When updating such balances, first `self.balances[tokenIndexFrom]` is updated and then `self.balances[tokenIndexTo]` is updated afterwards.

However when `tokenIndexFrom == tokenIndexTo` the second update overwrites the first update, causing token balances to be arbitrarily lowered. This also skews the exchange rates, allowing for swaps where value can be extracted.

Note: the protection against this problem is location in function `getY()`. However, this function is not called from `swapOut()`.

Note: the same issue exists in `swapInternalOut()`, which is called from `swapFromLocalAssetIfNeededForExactOut()` via `_swapAssetOut()`. However, via this route it is not possible to specify arbitrary token indexes. Therefore, there isn't an immediate risk here.

```
contract StableSwapFacet is BaseConnextFacet {
    ...
    function swapExactOut(... ,address assetIn, address assetOut, ... ) ... {
        return
            s.swapStorages[canonicalId].swapOut(
                   getSwapTokenIndex(canonicalId, assetIn),     // assetIn could be same as assetOut
                   getSwapTokenIndex(canonicalId, assetOut),
                   amountOut,
                   maxAmountIn
               );
    }
    ...
}
```

```
library SwapUtils {
    function swapOut(...,  uint8 tokenIndexFrom, uint8 tokenIndexTo, ... ) ... {
        ...
        uint256[] memory balances = self.balances;
        ...
        self.balances[tokenIndexFrom] = balances[tokenIndexFrom].add(dx).sub(dxAdminFee);
        self.balances[tokenIndexTo] = balances[tokenIndexTo].sub(dy); // overwrites previous update if
↪   From==To
        ...
    }
    function getY(..., uint8 tokenIndexFrom, uint8 tokenIndexTo, ... ) ... {
        ...
        require(tokenIndexFrom != tokenIndexTo, "compare token to itself"); // here is the protection
       ...
    }
}
```

Below is a proof of concept which shows that the `balances` of index `3` can be arbitrarily reduced.

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.14;
import "hardhat/console.sol";

contract test {
    uint[] balances = new uint[](10);
    function swap(uint8 tokenIndexFrom,uint8 tokenIndexTo,uint dx) public {
        uint dy=dx; // simplified
        uint256[] memory mbalances = balances;
        balances[tokenIndexFrom] = mbalances[tokenIndexFrom] + dx;
        balances[tokenIndexTo] = mbalances[tokenIndexTo] - dy;
    }
    constructor() {
        balances[3] = 100;
        swap(3,3,10);
        console.log(balances[3]); // 90
    }
}
```

Recommendation: Add the following to `swapExactOut()` and `swapInternalOut()`:

```
require(tokenIndexFrom != tokenIndexTo, "compare token to itself");
```

Connext: Solved in PR 1528.

Spearbit: Verified.

5.1.4 Use of spot price in `SponsorVault` leads to sandwich attack.

Severity: *Critical Risk*

Context: SponsorVault.sol#L208

Description: There is a special role sponsor in the protocol. Sponsors can cover the liquidity fee and transfer fee for users, making it more favorable for users to migrate to the new chain. Sponsors can either provide liquidity for each adopted token or provide the native token in the `SponsorVault`. If the native token is provided, the `SponsorVault` will swap to the adopted token before transferring it to users.

```
contract SponsorVault is ISponsorVault, ReentrancyGuard, Ownable {
  ...
  function reimburseLiquidityFees(
    address _token,
    uint256 _liquidityFee,
    address _receiver
  ) external override onlyConnext returns (uint256) {
  ...
  uint256 amountIn = tokenExchange.getInGivenExpectedOut(_token, _liquidityFee);
  amountIn = currentBalance >= amountIn ? amountIn : currentBalance;

  // sponsored fee may end being less than _liquidityFee due to slippage
  sponsoredFee = tokenExchange.swapExactIn{value: amountIn}(_token, msg.sender);
  ...
  }
}
```

The spot AMM price is used when doing the swap. Attackers can manipulate the value of `getInGivenExpectedOut` and make `SponsorVault` sell the native token at a bad price. By executing a sandwich attack the exploiters can drain all native tokens in the sponsor vault.

For the sake of the following example, assume that `_token` is `USDC` and native token is `ETH`, the sponsor tries to sponsor 100 `usdc` to the users:

- Attacker first manipulates the DEX and makes the exchange of 1 ETH = 0.1 USDC.
- `getInGivenExpectedOut` returns `100 / 0.1 = 1000`.
- `tokenExchange.swapExactIn` buys 100 `USDC` with 1000 `ETH`, causing the `ETH` price to decrease even lower.
- Attacker buys `ETH` at a lower prices and realizes a profit.

Recommendation: Instead of relying on DEXe's spot price the sponsor vault should rely instead on price quotes which are harder to manipulate, like those provided by an oracle (e.g. `chainlink` price, `uniswap` TWAP). The `SponsorVault` should fetch the oracle price and compare it against the spot price. The `SponsorVault` should either revert or use the oracle price when the spot price deviates from the oracle price.

Connext: Solved in PR 1595.

Spearbit: Verified.

5.2 High Risk

5.2.1 Configuration is crucial (both Nomad and Connext)

Severity: *High Risk*

Context: BridgeFacet.sol#L231-L238, BridgeFacet.sol#L257-L265, BridgeFacet.sol#L271-L276, Router.sol#L37-L39, XAppConnectionManager.sol#L106-L108, XAppConnectionManager.sol#L115-L125

Description: The Connext and Nomad protocol rely heavily on configuration parameters. These parameters are configured during deployment time and are updated afterwards. Configuration errors can have major consequences. Examples of important configurations are:

- BridgeFacet.sol: `s.promiseRouter`.
- BridgeFacet.sol: `s.connextions`.
- BridgeFacet.sol: `s.approvedSequencers`.
- Router.sol: `remotes[]`.
- xAppConnectionManager.sol: `home` .
- xAppConnectionManager.sol: `replicaToDomain[]`.
- xAppConnectionManager.sol: `domainToReplica[]`.

Recommendation: Have rigorous controls when configuring and updating these values.

5.2.2 Deriving price with `balanceOf` is dangerous

Severity: *High Risk*

Context: ConnextPriceOracle.sol#L109-L135

Description: `getPriceFromDex` derives the price by querying the balance of AMM's pools.

```
  function getPriceFromDex(address _tokenAddress) public view returns (uint256) {
    PriceInfo storage priceInfo = priceRecords[_tokenAddress];
    ...
    uint256 rawTokenAmount = IERC20Extended(priceInfo.token).balanceOf(priceInfo.lpToken);
    ...
    uint256 rawBaseTokenAmount = IERC20Extended(priceInfo.baseToken).balanceOf(priceInfo.lpToken);
     ...
}
```

Deriving the price with `balanceOf` is dangerous as `balanceOf` may be gamed. Consider univ2 as an example; Exploiters can first send tokens into the pool and pump the price, then absorb the tokens that were previously donated by calling mint.

Recommendation: Consider querying DEX's state through function calls such as Univ2's `getReserves()` which returns the correct state of the pool.

Connext: Solved in PR 1649.

Spearbit: Verified.

5.2.3 Routers can sybil attack the sponsor vault to drain funds

Severity: *High Risk*

Context: BridgeFacet.sol#L652-L688

Description: When funds are bridged from source to destination chain messages must first go through optimistic verification before being executed on the destination `BridgeFacet.sol` contract. Upon transfer processing the contract checks if the domain is sponsored. If such is the case then the user is reimbursed for both liquidity fees paid when the transfer was initiated and for the fees paid to the relayer during message propagation.

There currently isn't any mechanism to detect sybil attacks. Therefore, a router can perform several large value transfers in an effort to drain the sponsor vault of its funds. Because liquidity fees are paid to the router by a user connected to the router, there isn't any value lost in this type of attack.

Recommendation: Consider re-thinking the sponsor vault design or it may be safer to have it removed altogether.

Connext: Cap implemented in PR 1631. There is no total mitigation of sybil attacks on the vault possible, and this should be clearly explained to anyone who decides to deploy and fund one.

Spearbit: Verified and acknowledged.

5.2.4 Routers are exposed to extreme slippage if they attempt to repay debt before being reconciled

Severity: *High Risk*

Context: NomadFacet.sol#L188-L209, NomadFacet.sol#L269-L320, AssetLogic.sol#L228-L250, AssetLogic.sol#L308-L362

Description: When routers are reconciled, the local asset may need to be exchanged for the adopted asset in order to repay the unbacked Aave loan. `AssetLogic.swapFromLocalAssetIfNeededForExactOut()` takes two key arguments:

- `_amount` representing exactly how much of the adopted asset should be received.
- `_maxIn` which is used to limit slippage and limit how much of the local asset is used in the swap.

Upon failure to swap, the protocol will reset the values for unbacked Aave debt and distribute local tokens to the router. However, if this router partially paid off some of the unbacked Aave debt before being reconciled, `_maxIn` will diverge from `_amount`, allowing value to be extracted in the form of slippage. As a result, routers may receive less than the amount of liquidity they initially provided, leading to router insolvency.

Recommendation: Instead of using `_amount` to represent `_maxIn`, consider using some sort of user slippage amount. Alternatively, it may be easier/safer to restrict who can use Aave unbacked debt as there is a lot of added complexity in integrating unbacked debt into the protocol.

Connext: Solved in PR 1585.

Spearbit: Verified.

5.2.5 Malicious call data can DOS `execute`

Severity: *High Risk*

Context: Executor.sol#L142-L243

Description: An attacker can DOS the `executor` contract by giving `infinite` allowance to normal users. Since the `executor` increases allowance before triggering an external call, the tx will always revert if the allowance is already `infinite`.

```
function execute(ExecutorArgs memory _args) external payable override onlyConnext returns (bool, bytes
↪  memory) {
    ...
    if (!isNative && hasValue) {
      SafeERC20.safeIncreaseAllowance(IERC20(_args.assetId), _args.to, _args.amount); // reverts if set
↪  to `infinite` before
    }
    ...
    (success, returnData) = ExcessivelySafeCall.excessivelySafeCall(...) // can set to `infinite`
↪  allowance
    ...
}
```

Recommendation: Set the allowance to `0` before using `safeIncreaseAllowance`.

Note: also see issue `Not always safeApprove(..., 0)`

Connext: Solved in PR 1550.

Spearbit: Verified.

5.2.6 DOS attack on the Nomad `Home.sol` Contract

Severity: *High Risk*

Context: Home.sol#L332, Queue.sol#L119-L130

Description: Upon calling `xcall()`, a message is dispatched via Nomad. A hash of this message is inserted into the merkle tree and the new root will be added at the end of the queue. Whenever the updater of `Home.sol` commits to a new root, `improperUpdate()` will check that the new update is not fraudulent. In doing so, it must iterate through the queue of merkle roots to find the correct committed root. Because anyone can dispatch a message and insert a new root into the queue it is possible to impact the availability of the protocol by preventing honest messages from being included in the updated root.

```
function improperUpdate(..., bytes32 _newRoot, ... ) public notFailed returns (bool) {
    ...
    // if the _newRoot is not currently contained in the queue,
    // slash the Updater and set the contract to FAILED state
    if (!queue.contains(_newRoot)) {
        _fail();
        ...
    }
    ...
}

function contains(Queue storage _q, bytes32 _item) internal view returns (bool) {
    for (uint256 i = _q.first; i <= _q.last; i++) {
        if (_q.queue[i] == _item) {
            return true;
        }
    }
    return false;
}
```

Recommendation: Consider altering the queuing system such that `improperUpdate()` takes in an index argument that is greater than the old root. By specifying the index we can check that the new root is valid in O(1) time instead of O(n) time. Alternatively, it may be better to remove the queuing system altogether.

Connext: This is discussed in the Nomad Quantstamp audit report, and will be addressed by removing the queue for messaging in a future upgrade. Going to leave this issue open, though will note that this attack is costly to perform and (currently) exists within the Nomad protocol.

Spearbit: Acknowledged.

5.2.7 Upon failing to back unbacked debt `_reconcileProcessPortal()` will leave the converted asset in the contract

Severity: *High Risk*

Context: NomadFacet.sol#L225-L242

Description: When routers front liquidity for the protocol's users they are later reconciled once the bridge has optimistically verified transfers from the source chain. Upon being reconciled, the `_reconcileProcessPortal()` attempts to first pay back Aave debt before distributing the rest back to the router. However, `_reconcileProcessPortal()` will not convert the adopted asset back to the local asset in the case where the call to the Aave pool fails.

Instead, the function will set `amountIn = 0` and continue to distribute the local asset to the router.

```
if (success) {
  emit AavePortalRepayment(_transferId, adopted, backUnbackedAmount, portalFee);
} else {
  // Reset values
  s.portalDebt[_transferId] += backUnbackedAmount;
  s.portalFeeDebt[_transferId] += portalFee;

  // Decrease the allowance
  SafeERC20.safeDecreaseAllowance(IERC20(adopted), s.aavePool, totalRepayAmount);

  // Update the amount repaid to 0, so the amount is credited to the router
  amountIn = 0;
  emit AavePortalRepaymentDebt(_transferId, adopted, s.portalDebt[_transferId],
↪   s.portalFeeDebt[_transferId]);
}
```

Recommendation: It might be useful to convert the adopted asset amount back to the local asset such that subsequent swaps do not fail due to an insufficient amount of local asset. Alternatively, if the attempt to back unbacked debt fails, consider transferring the adopted asset out to the liquidity provider so they can handle this themselves.

Connext: We have decided to lean on the policy that Aave portals will not be automatically repaid.

Adding in the automatic repayment of portals adds complexity to the core codebase and leads to issues. Even with the portal repayment in place, issues such as a watcher disconnecting the `xapp` for an extended period mean we have to support out of band repayments regardless. By leaning on this as the only method of repayment, we are able to reduce the code complexity on reconcile.

Furthermore, it does not substantially reduce trust. Aave portals essentially amount to an unsecured credit line, usable by bridges. If the automatic repayment fails for any reason (i.e. due to bad pricing in the AMM), then the LP associated with the loan must be responsible for closing out the position in a trusted way.

Solved in PR 1585 by removing this code.

Spearbit: Verified.

5.2.8 `_handleExecuteTransaction()` doesn't handle native assets correctly

Severity: *High Risk*

Context: BridgeFacet.sol#L644-L718, Executor.sol#L142-L243

Description: The function `_handleExecuteTransaction()` sends any native tokens to the `executor` contract first, and then calls `s.executor.execute()`. This means that within that function `msg.value` will always be `0`. So the associated logic that uses `msg.value` doesn't work as expected and the function doesn't handle native assets correctly.

Note: also see issue "`Executor` reverts on receiving native tokens from `BridgeFacet`"

```
contract BridgeFacet is BaseConnextFacet {
    function _handleExecuteTransaction(...)... {
        ...
        AssetLogic.transferAssetFromContract(_asset, address(s.executor), _amount);
        (bool success, bytes memory returnData) = s.executor.execute(...); // no native tokens send
    }
}
contract Executor is IExecutor {
    function execute(ExecutorArgs memory _args) external payable override onlyConnext returns (bool,
↪   bytes memory) {
        ...
        if (isNative && msg.value != _args.amount) {  // msg.value is always 0
            ...
        }
    }
}
```

Recommendation: Change to code of `execute()` to handle previously send native assets. Or send the native assets along with the call to `execute()`.

Connext: Solved in PR 1532.

Spearbit: Verified.

Connext: Alternate approach: removed native asset handling. Implemented in PR 31.

Spearbit: Verified.

5.2.9 Add checks to `xcall()`

Severity: *High Risk*

Context: BridgeFacet.sol#L240-L339, BridgeFacet.sol#L400-L419, Executor.sol#L142-L280

Description: The function `xcall()` does some sanity checks, nevertheless more checks should be added to prevent issues later on in the use of the protocol.

If `_args.recovery== 0` then `_sendToRecovery()` will send funds to the `0` address, effectively losing them.

If `_params.agent == 0` the `forceReceiveLocal` can't be used and funds might be locked forever.

The `_args.params.destinationDomain` should never be `s.domain`, although this is also implicitly checked via `_mustHaveRemote()` assuming a correct configuration.

If `_args.params.slippageTol` is set to something greater than `s.LIQUIDITY_FEE_DENOMINATOR` then funds can be locked as `xcall()` allows for the user to provide the local asset, avoiding any swap while `_handleExecuteLiquidity()` in `execute()` may attempt to perform a swap on the destination chain.

```
function xcall(XCallArgs calldata _args) external payable nonReentrant whenNotPaused returns (bytes32) {
    // Sanity checks.
    ...
}
```

Recommendation: Consider adding the following checks:

- `recovery != 0`.
- `agent !=0`.
- `_args.params.destinationDomain != s.domain`.
- `_args.params.slippageTol <=s.LIQUIDITY_FEE_DENOMINATOR`.

Also doublecheck if any additional checks are useful.

Connext: Solved in PR 1536.

Spearbit: Verified.

5.2.10 `Executor` and `AssetLogic` deals with the native tokens inconsistently that breaks `execute()`

Severity: *High Risk*

Context: Executor.sol#L142 AssetLogic.sol#L127-L151, BridgeFacet.sol#L644-L718

Description: When dealing with an external callee the `BridgeFacet` will transfer liquidity to the `Executor` before calling `Executor.execute`.

In order to send the native token:

- The `Executor` checks for `_args.assetId == address(0)`.
- `AssetLogic.transferAssetFromContract()` disallows `address(0)`.

Note: also see issue *`Executor` reverts on receiving native tokens from `BridgeFacet`*.

```
contract BridgeFacet is BaseConnextFacet {
    function _handleExecuteTransaction() ...{
        ...
        AssetLogic.transferAssetFromContract(_asset, address(s.executor), _amount); // _asset may not
↪   be 0
        (bool success, bytes memory returnData) = s.executor.execute(
            IExecutor.ExecutorArgs(
                ...
              _asset,   // assetId parameter from ExecutorArgs  // must be 0 for Native asset
                ...
            )
          );
        ...
    }
}

library AssetLogic {
    function transferAssetFromContract( address _assetId, ... ) {
        ...
        // No native assets should ever be stored on this contract
        if (_assetId == address(0)) revert AssetLogic__transferAssetFromContract_notNative();

        if (_assetId == address(s.wrapper)) {
            // If dealing with wrapped assets, make sure they are properly unwrapped
            // before sending from contract
            s.wrapper.withdraw(_amount);
            Address.sendValue(payable(_to), _amount);
        }
    }
}

contract Executor is IExecutor {
    function execute(ExecutorArgs memory _args) external payable override onlyConnext returns (bool,
↪   bytes memory) {
        ...
        bool isNative = _args.assetId == address(0);
       ...
    }
}
```

The `BridgeFacet` cannot handle external callees when using native tokens.

Recommendation: The native tokens are either represented as `address(0)` or `address(wrapper)` throughout the whole code base, causing this inconsistency to be error prone. Recommend the team to go through the whole code base and make sure it's used consistently.

Connext: Solved in PR 1532.

Spearbit: Verified.

Connext: Alternate approach: removed native asset handling. Implemented in PR 1641.

Spearbit: Verified.

5.2.11 `Executor` reverts on receiving native tokens from `BridgeFacet`

Severity: *High Risk*

Context: Executor.sol BridgeFacet.sol#L696, AssetLogic.sol#L127-L151

Description: When doing an external call in `execute()`, the `BridgeFacet` provides liquidity into the Executor contract before calling `Executor.execute`. The `BridgeFacet` transfers native token when `address(wrapper)` is provided. The `Executor` however does not have a fallback/ receive function. Hence, the transaction will revert when the `BridgeFacet` tries to send the native token to the Executor contract.

```
function _handleExecuteTransaction(
    ...
    AssetLogic.transferAssetFromContract(_asset, address(s.executor), _amount);
    (bool success, bytes memory returnData) = s.executor.execute(...);
    ...
}
function transferAssetFromContract(...) ... {
    ...
    if (_assetId == address(s.wrapper)) {
        // If dealing with wrapped assets, make sure they are properly unwrapped
        // before sending from contract
        s.wrapper.withdraw(_amount);
        Address.sendValue(payable(_to), _amount);
    } else {
    ...
    }
}
```

Recommendation: Recommend to add a `receive` function in the `Executor` contract.

```
    receive() payable external {
        require(msg.sender == connext);
    }
```

Or unwrap the native asset and send it along with the call to the executor.

Connext: Ether sent along with the call. Solved in PR 1532.

Spearbit: Verified.

Connext: Alternate approach: removed native asset handling. Implemented in PR 31.

Spearbit: Verified.

5.2.12 `SponsorVault` sponsors full transfer amount in `reimburseLiquidityFees()`

Severity: *High Risk*

Context: BridgeFacet.sol#L660-L663

Description: The `BridgeFacet` passes `args.amount` as `_liquidityFee` when calling `reimburseLiquidityFees`. Instead of sponsoring `liquidityFee`, the sponsor vault would sponsor full transfer amount to the reciever.

Note: Luckily the amount in `reimburseLiquidityFees` is capped by `relayerFeeCap`.

```
function _handleExecuteTransaction(...) ... {
    ...
    (bool success, bytes memory data) = address(s.sponsorVault).call(
        abi.encodeWithSelector(s.sponsorVault.reimburseLiquidityFees.selector, _asset, _args.amount,
↪   _args.params.to)
    );
}
```

Recommendation: Pass `args.amount * (s.LIQUIDITY_FEE_DENOMINATOR - s.LIQUIDITY_FEE_NUMERATOR) / s.LIQUIDITY_FEE_DENOMINATOR` instead.

Connext: Solved in PR 1551.

Spearbit: Verified.

5.2.13 Tokens can get stuck in `Executor` contract if the `destination` doesn't claim them all

Severity: *High Risk*

Context: Executor.sol#L142-L243

Description: The function `execute()` increases allowance and then calls the recipient (`_args.to`). When the recipient does not use all tokens, these could remain stuck inside the `Executor` contract.

Note: the `executor` can have excess tokens, see: kovan executor. Note: see issue "Malicious call data can DOS `execute` or steal unclaimed tokens in the `Executor` contract".

```
function execute(...) ... {
    ...
    if (!isNative && hasValue) {
      SafeERC20.safeIncreaseAllowance(IERC20(_args.assetId), _args.to, _args.amount);
    }
    ...
    (success, returnData) = ExcessivelySafeCall.excessivelySafeCall( _args.to, ... );
    ...
}
```

Recommendation: Determine what should happen with unclaimed tokens. Consider one or more of the following suggestions:

- Send the unclaimed tokens to the recovery address via `_sendToRecovery()` (although this further complicates the contract).
- Set the `allowance` to `0` (before `safeIncreaseAllowance()` or after the call to `excessivelySafeCall()`).
- Allow the retrieval of unclaimed tokens from the `executor` contract by an owner.

Connext: New policy: "any funds left in the Executor following a transfer are claimable by anyone" . This forces implementers to think carefully about the calldata. Thus leave the issues as is.

Spearbit: Acknowledged.

Note: as it requires some deliberate action to retrieve the tokens, in practice several tokens will stay behind in the executor.

5.2.14 `reimburseLiquidityFees` send tokens twice

Severity: *High Risk*

Context: BridgeFacet.sol#L644-L675, SponsorVault.sol#L197-L226, ITokenExchange.sol#L18-L24

Description: The function `reimburseLiquidityFees()` is called from the `BridgeFacet`, making the `msg.sender` within this function to be `BridgeFacet`.

When using `tokenExchanges` via `swapExactIn()` tokens are sent to `msg.sender`, which is the `BridgeFacet`. Then, tokens are sent again to `msg.sender` via `safeTransfer()`, which is also the `BridgeFacet`.

Therefore, tokens end up being sent to the `BridgeFacet` twice.

Note: the check `...balanceOf(...) != starting + sponsored` should fail too.

Note: The fix in C4 seems to introduce this issue: code4rena-246

```
contract BridgeFacet is BaseConnextFacet {
     function _handleExecuteTransaction(... ) ... {
          ...
          uint256 starting = IERC20(_asset).balanceOf(address(this));
          ...
         (bool success, bytes memory data) = address(s.sponsorVault).call(
           abi.encodeWithSelector(s.sponsorVault.reimburseLiquidityFees.selector, _asset, _args.amount,
↪   _args.params.to)
         );
         if (success) {
             uint256 sponsored = abi.decode(data, (uint256));
             // Validate correct amounts are transferred
             if (IERC20(_asset).balanceOf(address(this)) != starting + sponsored) {   // this should
↪   fail now
                 revert BridgeFacet__handleExecuteTransaction_invalidSponsoredAmount();
             }
          ...
         }
         ...
    }
}
```

```
contract SponsorVault is ISponsorVault, ReentrancyGuard, Ownable {
    function reimburseLiquidityFees(... ) {
        if (address(tokenExchanges[_token]) != address(0)) {
            ...
            sponsoredFee = tokenExchange.swapExactIn{value: amountIn}(_token, msg.sender);  // send to
↪   msg.sender
        } else {
            ...
        }
        ...
        IERC20(_token).safeTransfer(msg.sender, sponsoredFee); // send again to msg.sender
    }
}
```

```
interface ITokenExchange {
    /**
    * @notice Swaps the exact amount of native token being sent for a given token.
    * @param token The token to receive
    * @param recipient The recipient of the token
    * @return The amount of tokens resulting from the swap
    */
    function swapExactIn(address token, address recipient) external payable returns (uint256);
}
```

Recommendation: Doublecheck the code to see what the intended behavior is.

Connext: Solved in PR 1551.

Spearbit: Verified.

5.2.15 Anyone can repay the `portalDebt` with different tokens

Severity: *High Risk*

Context: PortalFacet.sol#L80-L113 PortalFacet.sol#L115-L167

Description: Routers can provide liquidity in the protocol to improve the UX of cross-chain transfers. Liquidity is sent to users under the router's consent before the cross-chain message is settled on the optimistic message protocol, i.e., Nomad. The router can also borrow liquidity from `AAVE` if the router does not have enough of it. It is the router's responsibility to repay the debt to AAVE.

```
contract PortalFacet is BaseConnextFacet {
  function repayAavePortalFor(
    address _adopted,
    uint256 _backingAmount,
    uint256 _feeAmount,
    bytes32 _transferId
  ) external payable {
    address adopted = _adopted == address(0) ? address(s.wrapper) : _adopted;
    ...
    // Transfer funds to the contract
    uint256 total = _backingAmount + _feeAmount;
    if (total == 0) revert PortalFacet__repayAavePortalFor_zeroAmount();

    (, uint256 amount) = AssetLogic.handleIncomingAsset(_adopted, total, 0);
    ...
    // repay the loan
    _backLoan(adopted, _backingAmount, _feeAmount, _transferId);
  }
}
```

The `PortalFacet` does not check whether `_adopted` is the correct token in debt. Assume that the protocol borrows `ETH` for the current `_transferId`, therefore `Router` should repay `ETH` to clear the debt. However, the `Router` can provide any valid tokens, e.g. `DAI`, `USDC`, to clear the debt. This results in the insolvency of the protocol.

Note: a similar issue is also present in `repayAavePortal()`.

Recommendation: Check `_adopted` is the correct token in this transfer.

Connext: Solved in PR 1559.

Spearbit: Verified.

5.2.16 Malicious call data can steal unclaimed tokens in the `Executor` contract

Severity: *High Risk*

Context: Executor.sol#L211

Description: Users can provide a destination contract `args.to` and arbitrary data `_args.callData` when doing a cross-chain transfer. The protocol will provide the allowance to the callee contract and triggers the function call through `ExcessivelySafeCall.excessivelySafeCall`.

```
contract Executor is IExecutor {
    function execute(ExecutorArgs memory _args) external payable override onlyConnext returns (bool,
↪   bytes memory) {
        ...
        SafeERC20.safeIncreaseAllowance(IERC20(_args.assetId), _args.to, _args.amount);
        ...

        // Try to execute the callData
        // the low level call will return `false` if its execution reverts
        (success, returnData) = ExcessivelySafeCall.excessivelySafeCall(
        _args.to,
        gas,
        isNative ? _args.amount : 0,
        MAX_COPY,
        _args.callData
        );
        ...
    }
}
```

Since there aren't restrictions on the destination contract and calldata, exploiters can steal the tokens from the `executor`.

Note: the executor does have excess tokens, see: see: kovan executor.

Note: see issue *Tokens can get stuck in Executor contract*.

Tokens can be stolen by granting an allowance. Setting

```
calldata = abi.encodeWithSelector(ERC20.approve.selector, exploiter, type(uint256).max);
```

and `args.to = tokenAddress` allows the exploiter to get an infinite allowance of any token, effectively stealing any unclaimed tokens left in the `executor`.

Recommendation: The protocol could communicate with the callee contract through a callback function. A possible specification of the callback:

```
function connextExecute(uint32 origin, address adoptedToken, address originSender, uint256 amount,
↪   bytes calldata callData) returns(bytes4)
```

This results in higher gas efficiency because callees do not have to query `origin`, `originSender`, and `amount` through three separate external calls.

Note: this way arbitrary calls are not possible anymore.

Connext: New policy: "any funds left in the Executor following a transfer are claimable by anyone". This forces implementers to think carefully about the calldata. Thus leave the issues as is.

Spearbit: Acknowledged.

5.3 Medium Risk

5.3.1 Fee-On-Transfer tokens are not explicitly denied in `swap()`

Severity: *Medium Risk*

Context: SwapUtils.sol#L690-L729

Description: The `swap()` function is used extensively within the Connext protocol, primarily when swapping between local and adopted assets. When a swap is performed, the function will check the actual amount transferred. However, this is not consistent with other swap functions which check that the amount transferred is equal to `dx`. As a result, overwriting `dx` with `tokenFrom.balanceOf(address(this)).sub(beforeBalance)` allows for fee-on-transfer tokens to work as intended.

Recommendation: Consider adding a `require(dx == tokenFrom.balanceOf(address(this)).sub(beforeBalance), "not support fee token");` check prior to overwriting `dx` to ensure fee-on-transfer tokens are not used in the swap.

Connext: Solved in PR 1642, in this commit.

Spearbit: Verified.

5.3.2 `xcall()` may erroneously overwrite prior calls to `bumpTransfer()`

Severity: *Medium Risk*

Context: BridgeFacet.sol#L380-L386, BridgeFacet.sol#L313

Description: The `bumpTransfer()` function allows users to increment the relayer fee on any given `transferId` without checking if the unique transfer identifier exists. As a result, a subsequent call to `xcall()` will overwrite the `s.relayerFees` mapping, leading to lost funds.

Recommendation: Consider adding a check in `bumpTransfer()` to ensure `_transferId` exists. This mitigation can be implemented in a similar fashion to `PromiseRouter.bumpCallbackFee()`. It is important to note that checking for a non-zero `s.relayerFees` is not sufficient as `xcall()` accepts a zero values. Alternatively, it may be more succinct to modify `xcall()` such that `s.relayerFees` is incremented instead of overridden.

Connext: Solved in PR 1643.

Spearbit: Verified.

Note: remaining risk `bumpTransfer()` allow adding funds to an invalid `transferId`. This is comparable to transferring tokens to the wrong address.

5.3.3 `_handleExecuteLiquidity` doesn't consistently check for `receiveLocalOverrides`

Severity: *Medium Risk*

Context: BridgeFacet.sol#L571-L638

Description: The function `_handleExecuteLiquidity()` initially checks for `receiveLocal` but does not check for `receiveLocalOverrides`. Later on it does check for both of values.

```
function _handleExecuteLiquidity(... ) ... {
    ...
        if (
          !_args.params.receiveLocal && // doesn't check for receiveLocalOverrides
          s.routerBalances[_args.routers[0]][_args.local] < toSwap &&
          s.aavePool != address(0)
        ) {
    ...
    if (_args.params.receiveLocal || s.receiveLocalOverrides[_transferId]) { // extra check
      return (toSwap, _args.local);
    }
}
```

As a result, the portal may pay the bridge user in the adopted asset when they opted to override this behaviour to avoid slippage conditions outside of their boundaries, potentially leading to an unwarranted reception of funds denominated in the adopted asset.

Recommendation: Consider adding a check for `receiveLocalOverrides` to the Aave portal eligibility check.

Connext: Solved in PR 1644.

Spearbit: Verified.

5.3.4 Router signatures can be replayed when executing messages on the destination domain

Severity: *Medium Risk*

Context: BridgeFacet.sol#L476-496

Description: Connext bridge supports near-instant transfers by allowing users to pay a small fee to routers for providing them with liquidity. Gelato relayers are tasked with taking in bids from liquidity providers who sign a message consisting of the `transferId` and path length. The path length variable only guarantees that the message they signed will only be valid if `_args.routers.length - 1` routers are also selected. However, it does not prevent Gelato relayers from re-using the same signature multiple times. As a result, routers may unintentionally provide more liquidity than expected.

Recommendation: Consider ensuring that a router's signed message can only be used once for a given `transferId`. It may be useful to track these in a boolean mapping.

Connext: Solved in PR 1626.

Spearbit: Verified.

Note: this still assumes that the sequencer is a centralized role maintained by the Connext team. We understand that this will be addressed in future on-chain changes to incentivize honest behavior and further decentralize the sequencer role.

Connext: Currently the sequencer is a centralized role, and will be decentralized in the future.

Consider that the only 'attack vector' here (really more of a griefing vector) is that the sequencer has only the potential to favor certain routers over others, and cannot steal anyone's funds. Additionally, we know that the 'randomness' of the sequencer selection - while not strictly enforceable on-chain - will at the very least be demonstrated publicly; anyone can check to see that our sequencer has been behaving politely (simply check the distribution of router-usage over time, it should be relatively even). So it should be okay to continue this in a centralized manner for the time being, since funds are not jeopardized, and the only trust vector here is that we continue to select routers fairly to make sure everyone gets a fair share of profits.

For clarity's sake: the model towards decentralization will probably involve 'fair selection' being enforceable through staking/slashing in the future.

5.3.5 `diamondCut()` allows re-execution of old updates

Severity: *Medium Risk*

Context: LibDiamond.sol#L95-L119

Description: The function `diamondCut()` of `LibDiamond` verifies the signed version of the update parameters. It checks the signed version is available and a sufficient amount of time has passed. However it doesn't prevent multiple executions and the signed version stays valid forever.

This allows old updates to be executed again. Assume the following:

- facet_x (or function_y) has value: `version_1`.
- then: `replace facet_x (or function_y) with version_2`.
- then a bug is found in `version_2` and it is rolled back with: `replace facet_x (or function_y) with version_1`.

- then a (malicious) owner could immediately do: `replace facet_x (or function_y) with version_2` (because it is still valid).

Note: the risk is limited because it can only executed by the contract owner, however this is probably not how the mechanism should work.

```
library LibDiamond {
    function diamondCut(...) ... {
        ...
        uint256 time = ds.acceptanceTimes[keccak256(abi.encode(_diamondCut, _init, _calldata))];
        require(time != 0 && time < block.timestamp, "LibDiamond: delay not elapsed");
        ...
    }
}
```

Recommendation: Consider doing the following:

- Add a validity period for updates;
- Remember which updates have been executed and prevent re-execution;
- Add a nonce (for cases where a re-execution is wanted).

Connext: Solved in PR 1576.

Spearbit: Verified.

5.3.6 Not always `safeApprove(..., 0)`

Severity: *Medium Risk*

Context: NomadFacet.sol#L176-L242, AssetLogic.sol#L308-L362, PortalFacet.sol#L179-L197, AssetLogic.sol#L263-L295, Executor.sol#L142-L339

Description: Some functions like `_reconcileProcessPortal` of `BaseConnextFacet` and `_swapAssetOut` of `AssetLogic` do `safeApprove(..., 0)` first.

```
contract NomadFacet is BaseConnextFacet {
    function _reconcileProcessPortal( ... ) ... {
        ...
        // Edge case with some tokens: Example USDT in ETH Mainnet, after the backUnbacked call there could be a remaining allowance if not the whole amount is pulled by aave.
        // Later, if we try to increase the allowance it will fail. USDT demands if allowance is not 0, it has to be set to 0 first.
        // Example:
        [ParaSwapRepayAdapter.sol#L138-L140](https://github.com/aave/aave-v3-periphery/blob/ca184e5278bcbc10d28c3dbbc604041d7cfac50b/contracts/adapters/paraswap/ParaSwapRepayAdapter.sol#L138-L140)
        SafeERC20.safeApprove(IERC20(adopted), s.aavePool, 0);
        SafeERC20.safeIncreaseAllowance(IERC20(adopted), s.aavePool, totalRepayAmount);
        ...
    }
}
```

While the following functions don't do this:

- `xcall` of `BridgeFacet`.
- `_backLoan` of `PortalFacet`.
- `_swapAsset` of `AssetLogic`.
- `execute` of `Executor`.

This could result in problems with tokens like `USDT`.

```
contract BridgeFacet is BaseConnextFacet {
    function xcall(XCallArgs calldata _args) external payable nonReentrant whenNotPaused returns
↪   (bytes32) {
        ...
        SafeERC20.safeIncreaseAllowance(IERC20(bridged), address(s.bridgeRouter), bridgedAmt);
        ...
    }
}
contract PortalFacet is BaseConnextFacet {
    function _backLoan(...) ... {
        ...
        SafeERC20Upgradeable.safeIncreaseAllowance(IERC20Upgradeable(_asset), s.aavePool, _backing +
↪   _fee);
        ...
    }
}
library AssetLogic {
    function _swapAsset(...) ... {
        ...
        SafeERC20.safeIncreaseAllowance(IERC20(_assetIn), address(pool), _amount);
        ...
    }
}
contract Executor is IExecutor {
    function execute( ... ) ... {
        ...
        SafeERC20.safeIncreaseAllowance(IERC20(_args.assetId), _args.to, _args.amount);
        ...
    }
}
```

Recommendation: Consider adding `safeApprove(..., 0)`.

Connext: Solved in PR 1550.

Spearbit: Verified.

5.3.7 `_slippageTol` does not adjust for decimal differences

Severity: *Medium Risk*

Context: AssetLogic.sol#L273

Description: Users set the slippage tolerance in percentage. The `assetLogic` calculates:

```
minReceived =  (_amount * _slippageTol) / s.LIQUIDITY_FEE_DENOMINATOR
```

Then `assetLogic` uses `minReceived` in the swap functions. The `minReceived`, however, does not adjust for the decimal differences between `assetIn` and `assetOut`. Users will either always hit the slippage or suffer huge slippage when `assetIn` and `assetOut` have a different number of decimals.

Assume the number of decimals of `assetIn` is 6 and the decimal of `assetOut` is 18. The `minReceived` will be set to `10^-12` smaller than the correct value. Users would be vulnerable to sandwich attacks in this case. Assume the number of decimals of `assetIn` is 18 and the number of decimals of `assetOut` is 6. The `minReceived` will be set to `10^12` larger than the correct value. Users would always hit the slippage and the cross-chain transfer will get stuck.

```
library AssetLogic {
    function _swapAsset(... ) ... {
        // Swap the asset to the proper local asset
        uint256 minReceived = (_amount * _slippageTol) / s.LIQUIDITY_FEE_DENOMINATOR;
        ...
        return (pool.swapExact(_amount, _assetIn, _assetOut, minReceived), _assetOut);
        ...
    }
}
```

Recommendation: Recommend to adjust the value with `swapStorage.tokenPrecisionMultipliers` for internal swap. For the external swap, the value should be adjusted according to `token.decimals`.

Connext: Solved in PR 1574.

Spearbit: Verified.

5.3.8 Canonical assets should be keyed on the hash of domain and id

Severity: *Medium Risk*

Context: LibConnextStorage.sol#L184, AssetLogic.sol#L36, AssetFacet.sol#L143, TokenRegistry.sol#L112-L113, TokenRegistry.sol#L334

Description: A canonical asset is a tuple of a `(domain, id)` pair. `TokenRegistry`'s owner has the power to register new tokens in the system (See `TokenRegistry.ensureLocalToken()` and `TokenRegistry.enrollCustom()`). A canonical asset is registered using the hash of its domain and id (See `TokenRegistry._setCanonicalToRepresentation()`).

Connext uses only the id of a canonical asset to uniquely identify. Here are a few references:

- `swapStorages`
- `canonicalToAdopted`

It is an issue if `TokenRegistry` registers two canonical assets with the same id. If this id fetches the incorrect canonical asset an unintended one might be transferred to the destination chain, of the transfers may revert.

Recommendation: Consider using the `keccak256` hash of canonical asset's domain and id for mapping keys.

Connext: Solved in PR 1588.

Spearbit: Verified.

5.3.9 Missing checks for Chainlink oracle

Severity: *Medium Risk*

Context: ConnextPriceOracle.sol#L98, ConnextPriceOracle.sol#L153

Description: `ConnextPriceOracle.getTokenPrice()` function goes through a series of oracles. At each step, it has a few validations to avoid incorrect price. If such validations succeed, the function returns the non-zero oracle price. For the Chainlink oracle, `getTokenPrice()` ultimately calls `getPriceFromChainlink()` which has the following validation —

```
if (answer == 0 || answeredInRound < roundId || updateAt == 0) {
  // answeredInRound > roundId ===> ChainLink Error: Stale price
  // updatedAt = 0 ===> ChainLink Error: Round not complete
  return 0;
}
```

`updateAt` refers to the timestamp of the round. This value isn't checked to make sure it is recent.

Additionally, it is important to be aware of the `minAnswer` and `maxAnswer` of the Chainlink oracle, these values are not allowed to be reached or surpassed. See Chainlink API reference for documentation on `minAnswer` and `maxAnswer` as well as this piece of code: OffchainAggregator.sol

Recommendation:

- Determine the tolerance threshold for `updateAt`. If `block.timestamp - updateAt` exceeds that threshold, return 0 which is consistent with how the current validations are handled.
- Consider having off-chain monitoring to identify when the market price moves out of `[minAnswer, maxAnswer]` range.

Connext: Recency check is implemented in PR 1602. Off chain monitoring will be considered.

Spearbit: Verified and acknowledged.

5.3.10 Same `params.SlippageTol` is used in two different swaps

Severity: *Medium Risk*

Context: BridgeFacet.sol#L299-L304 BridgeFacet.sol#L637

Description: The Connext protocol does a cross-chain transfer with the help of the Nomad protocol. In order to use the Nomad protocol, Connext has to convert the adopted token into the local token. For a cross-chain transfer, users take up two swaps. `Adopted -> Local at the source chain` and `Local -> Adopted at the destination chain`.

BridgeFacet.sol#L299-L304

```
function xcall(XCallArgs calldata _args) external payable whenNotPaused nonReentrant returns (bytes32) {
    ...
      // Swap to the local asset from adopted if applicable.
      (uint256 bridgedAmt, address bridged) = AssetLogic.swapToLocalAssetIfNeeded(
        canonical,
        transactingAssetId,
        amount,
        _args.params.slippageTol
    );
    ...
}
```

BridgeFacet.sol#L637

```
  function _handleExecuteLiquidity(
    bytes32 _transferId,
    bytes32 _canonicalId,
    bool _isFast,
    ExecuteArgs calldata _args
  ) private returns (uint256, address) {
    ...
    // swap out of mad* asset into adopted asset if needed
    return AssetLogic.swapFromLocalAssetIfNeeded(_canonicalId, _args.local, toSwap,
↪   _args.params.slippageTol);
  }
```

The same slippage tolerance `_args.params.slippageTol` is used in two swaps. In most cases users cannot set the correct slippage tolerance to protect two swaps.

Assume the Nomad asset is slightly cheaper in both chains. 1 Nomad asset equals 1.01 adopted asset. An expected swap would be:`1 adopted -> 1.01 Nomad asset -> 1 adopted`. The right slippage tolerance should be set at 1.01 and 0.98 respectively. Users cannot set the correct tolerance with a single parameter. This makes users vulnerable to MEV searchers. Also, user transfers get stuck during periods of instability.

Recommendation: Allow users to set two different slippage tolerance for the two swaps.

Connext: Solved in PR 1575.

Spearbit: Verified.

5.4 Low Risk

5.4.1 `getTokenPrice()` returns stale token prices

Severity: *Low Risk*

Context: ConnextPriceOracle.sol#L88-L107

Description: `getTokenPrice()` reads from the `assetPrices[tokenAddress].price` mapping which stores the latest price as configured by the protocol admin in `setDirectPrice()`. However, the check for a stale token price will never fallback to other price oracles as `tokenPrice != 0`. Therefore, the stale token price will be unintentionally returned.

Recommendation: If `assetPrices[tokenAddress].updatedAt` is considered stale, consider setting `tokenPrice` to zero such that the function attempts to query the fallback oracles.

Connext: Solved in PR 1647.

Spearbit: Verified.

5.4.2 Potential division by zero if gas token oracle is faulty

Severity: *Low Risk*

Context: SponsorVault.sol#L250-L252

Description: In the event that the gas token oracle is faulty and returns malformed values, the call to `reimburseRelayerFees()` in `_handleExecuteTransaction()` will fail. Fortunately, the low-level `call()` function will not prevent the transfer from being executed, however, this may lead to further issues down the line if changes are made to the sponsor vault.

Recommendation: Consider checking that `den != 0` before calculating `sponsoredFee`.

Connext: Solved in PR 1645.

Spearbit: Verified.

5.4.3 Burn does not lower allowance

Severity: *Low Risk*

Context: BridgeRouter.sol#L252-L280, BridgeToken.sol#L62-L64

Description: The function `_takeTokens()` of `BridgeRouter` takes in the tokens from the sender. Sometimes it transfers them and sometimes it burns them. In the case of burning the tokens, the allowance isn't "used up".

```
function _takeTokens(... ) ... {
     ...
     if (tokenRegistry.isLocalOrigin(_token)) {
        ...
        IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
        ...
    } else {
        ...
        _t.burn(msg.sender, _amount);    // doesn't use up the allowance
        ...
    }
    ...
}
contract BridgeToken is Version0, IBridgeToken, OwnableUpgradeable, ERC20 {
    ...
    function burn(address _from, uint256 _amnt) external override onlyOwner {
        _burn(_from, _amnt);
    }
}
```

Recommendation: Consider using a function like the OZ `BurnFrom()` which does use up the allowance.

Connext: The BridgeRouter basically has unlimited allowance for BridgeTokens, by design. The user doesn't allow the Bridge to burn tokens, and so there is no allowance to reduce.

Spearbit: Acknowledged

5.4.4 Two step ownership transfer

Severity: *Low Risk*

Context: ConnextPriceOracle.sol#L221-L226, BridgeRouter.sol#L479-L486

Description: The function `setAdmin()` transfer ownership to a new address. In case a wrong address is supplied ownership is inaccessible. The same issue occurs with `transferOwnership` of `OwnableUpgradeable` in several Nomad contracts. Additionally the Nomad contract try to prevent `renounceOwnership`, however, this can also be accomplished with `transferOwnership` to a non existing address.

Relevant Nomad contracts:

- `TokenRegistry.sol`
- `NomadBase.sol`
- `UpdaterManager.sol`
- `XAppConnectionManager.sol`

```
contract ConnextPriceOracle is PriceOracle {
    ...
    function setAdmin(address newAdmin) external onlyAdmin {
        address oldAdmin = admin;
        admin = newAdmin;
        emit NewAdmin(oldAdmin, newAdmin);
   }
}

contract BridgeRouter is Version0, Router {
    ...
    /**
     * @dev should be impossible to renounce ownership;
     *      we override OpenZeppelin OwnableUpgradeable's
     *      implementation of renounceOwnership to make it a no-op
     */
    function renounceOwnership() public override onlyOwner {
        // do nothing
    }
}
```

Recommendation: Consider implementing a two step ownership transfer.

Connext: Will not change for `BridgeRouter`. Solved for the `PriceOracle` in PR 1605.

Spearbit: Acknowledged and verified.

5.4.5 Function `removeRouter` does not clear `approvedForPortalRouters`

Severity: *Low Risk*

Context: RoutersFacet.sol#L293-L325, LibConnextStorage.sol#L104-L111

Description: The function `removeRouter()` clears most of the fields of the struct `RouterPermissionsManagerInfo` except for `approvedForPortalRouters`.

However, it is still good to also remove `approvedForPortalRouters` in `removeRouter()` because if the router were to be added again later (via `setupRouter()`) or `_isRouterOwnershipRenounced` is set in the future, the router would still have the old `approvedForPortalRouters`.

```
struct RouterPermissionsManagerInfo {
  mapping(address => bool) approvedRouters; // deleted
  mapping(address => bool) approvedForPortalRouters; // not deleted
  mapping(address => address) routerRecipients; // deleted
  mapping(address => address) routerOwners; // deleted
  mapping(address => address) proposedRouterOwners; // deleted
  mapping(address => uint256) proposedRouterTimestamp; // deleted
}

contract RoutersFacet is BaseConnextFacet {
    function removeRouter(address router) external onlyOwner {
        ...
        s.routerPermissionInfo.approvedRouters[router] = false;
        ...
        s.routerPermissionInfo.routerOwners[router] = address(0);
        ...
        s.routerPermissionInfo.routerRecipients[router] = address(0);
        ...
        delete s.routerPermissionInfo.proposedRouterOwners[router];
        delete s.routerPermissionInfo.proposedRouterTimestamp[router];
    }
}
```

Recommendation: In function `removeRouter` also clear `approvedForPortalRouters`.

Connext: Solved in PR 1586.

Spearbit: Verified.

5.4.6 Anyone can self burn lp token of the AMM

Severity: *Low Risk*

Context: LPToken.sol

Description: When providing liquidity into the AMM pool, users get LP tokens. Users can redeem their shares of the liquidity by redeeming LP to the AMM pool.

The current `LPToken` contract inherits Openzepplin's `ERC20BurnableUpgradeable`. Users can burn their tokens by calling `burn` without notifying the AMM pools. ERC20BurnableUpgradeable.sol#L26-L28. Although users do not profit from this action, it brings up concerns such as:

- An exploiter has an easy way to pump the LP price. Burning LP is similar to donating value to the pool. While it's good for the pool, this gives the exploiter another tool to break other protocols. After the cream finance attack many protocols started to take extra caution and made this a restricted function (absorbing donation) github.com/yearn/yearn-security/blob/master/disclosures/2021-10-27.md.
- Against the best practice. Every state of an AMM is related to price. Allowing external actors to change the AMM states without notifying the main contract is dangerous. It's also harder for a developer to build other novel AMM based on the same architecture.

Note: the `burn` function is also not protected by `nonReentrant` or `whenNotPaused`.

Recommendation: Add an `onlyOwner` modifier. Only the AMM pool should be able to burn LP tokens.

Connext: Solved in PR 1606.

Spearbit: Verified.

5.4.7 Skip timeout in `diamondCut()` (edge case)

Severity: *Low Risk*

Context: LibDiamond.sol#L95-L119

Description: Edge case: If someone manages to get an update through which deletes all facets then the next update skips the delay (because `ds.facetAddresses.length` will be 0).

```
library LibDiamond {
    function diamondCut(...) ... {
        ...
        if (ds.facetAddresses.length != 0) {
            uint256 time = ds.acceptanceTimes[keccak256(abi.encode(_diamondCut, _init, _calldata))];
            require(time != 0 && time < block.timestamp, "LibDiamond: delay not elapsed");
        } // Otherwise, this is the first instance of deployment and it can be set automatically
        ...
    }
}
```

Recommendation: Make it so the skipping of a timeout can only be done once, for example by setting a flag.

Connext: Solved in PR 1607.

Spearbit: Verified.

5.4.8 Limit gas for `s.executor.execute()`

Severity: *Low Risk*

Context: BridgeFacet.sol#L644-L718

Description: The call to `s.executor.execute()` in `BridgeFacet` might use up all available gas. In that case, the call to `callback` to report to the originator might not be called because the execution stops due an out of gas error.

Note: the `execute()` function might be retried by the `relayer` so perhaps this will fix itself eventually.

Note: `excessivelySafeCall` in `Executor` does limit the amount of gas.

```
contract BridgeFacet is BaseConnextFacet {
    function _handleExecuteTransaction(...) ... {
        ...
        (bool success, bytes memory returnData) = s.executor.execute(...); // might use all available
↪   gas
        ...
        // If callback address is not zero, send on the PromiseRouter
        if (_args.params.callback != address(0)) {
            s.promiseRouter.send(...);   // might not have enough gas
        }
        ...
    }
}
```

Recommendation: Consider limiting the amount of gas sent to `s.executor.execute()`.

Connext: It's the responsibility of the relayer to do a proper gas estimate (and leave a bit of overhead). The user should have provided a sufficient fee (in ETH) on the sending chain for the relayer to be slightly generous in its gas estimation. If the call reverts (for any reason, including out of gas), the relayer can just call it again (perhaps pending another bump from the user in their gas fee).

Spearbit: Acknowledged.

5.4.9 Several `external` functions missing `whenNotPaused` mofifier

Severity: *Low Risk*

Context: BridgeFacet.sol#L380-L386, PortalFacet.sol#L80-L167

Description: The following functions don't have a `whenNotPaused` modifier while most other external functions do.

- `bumpTransfer` of `BridgeFacet`.
- `forceReceiveLocal` of `BridgeFacet`.
- `repayAavePortal` of `PortalFacet`.
- `repayAavePortalFor` of `PortalFacet`.

Without `whenNotPaused` these functions can still be executed when the protocol is paused.

Recommendation: Doublecheck to see if `whenNotPaused` useful.

Connext: Given earlier conclusions about portal repayment being taken out-of-band, we're going to acknowledge but not fix `repayAavePortal` and `repayAavePortalFor`.

`bumpTransfer` fixed (`whenNotPaused` was added) in PR 1377.

`forceReceiveLocal` only modifies the state property to override whether to receive local token, and it's permissioned. Doesn't seem necessary to include `whenNotPaused` for this reason. Acknowledged on that front.

Spearbit: Verified and acknowledged.

5.4.10 Gas griefing attack on callback execution

Severity: *Low Risk*

Context: PromiseRouter.sol#L250

Description: When the callback is executed on the source chain the following line can revert or consume all forwarded gas. In this case, the relayer wastes gas and doesn't get the callback fee.

```
ICallback(callbackAddress).callback(transferId, _msg.returnSuccess(), _msg.returnData());
```

Recommendation:

- Decide on a fixed gas stipend to forward to this call, so that even if it consumes all of it, the transaction still has enough to continue.
- Enclose the call in a try-catch block. In case of a revert from callback, continue the execution and transfer the callback fee to `msg.sender`.

Connext: Solved in PR 1610.

Spearbit: Verified.

5.4.11 Callback fails when `returnData` is empty

Severity: *Low Risk*

Context: PromiseRouter.sol#L173

Description: If a transfer involves a callback, `PromiseRouter` reverts if `returnData` is empty.

```
if (_returnData.length == 0) revert PromiseRouter__send_returndataEmpty();
```

However, the callback should be allowed in case the user wants to report the calldata execution success on the destination chain (`_returnSuccess`).

Recommendation: Consider removing the revert when `_returnData` is empty. Delete the line at PromiseRouter.sol#L173.

Connext: Solved in PR 1587.

Spearbit: Verified.

5.5 Gas Optimization

5.5.1 Redundant fee on transfer logic

Severity: *Gas Optimization*

Context: PortalFacet.sol#L124-L167, AssetLogic.sol#L66-L90, AssetLogic.sol#L108-L118

Description: The function `repayAavePortalFor()` has logic for fee on transfer tokens. However, `handleIncomingAsset()` doesn't allow fee on transfer tokens. So this extra code shouldn't be necessary in `repayAavePortalFor()`.

```
function repayAavePortalFor(...) ... {
    ...
    (, uint256 amount) = AssetLogic.handleIncomingAsset(_adopted, total, 0);
    ...
    // If this was a fee on transfer token, reduce the total
    if (amount < total) {
      uint256 missing;
      unchecked {
        missing = total - amount;
      }
      if (missing < _feeAmount) {
        // Debit fee amount
        unchecked {
          _feeAmount -= missing;
        }
      } else {
        // Debit backing amount
        unchecked {
          missing -= _feeAmount;
        }
        _feeAmount = 0;
        _backingAmount -= missing;
      }
    }
   ...
}
```

```
library AssetLogic {
    function handleIncomingAsset(...) ... {
        ...
        // Transfer asset to contract
        trueAmount = transferAssetToContract(_assetId, _assetAmount);
          ....
    }
    function transferAssetToContract(address _assetId, uint256 _amount) internal returns (uint256) {
        ...
        // Validate correct amounts are transferred
        uint256 starting = IERC20(_assetId).balanceOf(address(this));
        SafeERC20.safeTransferFrom(IERC20(_assetId), msg.sender, address(this), _amount);
        // Ensure this was not a fee-on-transfer token
        if (IERC20(_assetId).balanceOf(address(this)) - starting != _amount) {
            revert AssetLogic__transferAssetToContract_feeOnTransferNotSupported();
        }
        ...
    }
}
```

Recommendation: Double check the fee on transfer handling.

Connext: Solved in PR 1550.

Spearbit: Verified.

5.5.2 Some gas can be saved in `reimburseLiquidityFees`

Severity: *Gas Optimization*

Context: SponsorVault.sol#L197-L226

Description: Some gas can be saved by assigning `tokenExchange` before the `if` statement. This also improves readability.

```
function reimburseLiquidityFees(...) ... {
    ...
    if (address(tokenExchanges[_token]) != address(0)) {  // could use `tokenExchange`

      ITokenExchange tokenExchange = tokenExchanges[_token]; // do before the if
}
```

Recommendation: Assign `tokenExchange` before the `if` statement.

Connext: Solved in PR 1654.

Spearbit: Verified.

5.5.3 `LIQUIDITY_FEE_DENOMINATOR` could be a constant

Severity: *Gas Optimization*

Context: BridgeFacet.sol, PortalFacet.sol, AssetLogic.sol

Description: The value of `LIQUIDITY_FEE_DENOMINATOR` seems to be constant. However, it is currently stored in `s` and requires an `SLOAD` operation to retrieve it, increasing gas costs.

```
upgrade-initializers/DiamondInit.sol:      s.LIQUIDITY_FEE_DENOMINATOR = 10000;
BridgeFacet.sol:      toSwap = _getFastTransferAmount(..., s.LIQUIDITY_FEE_DENOMINATOR);
BridgeFacet.sol:    s.portalFeeDebt[_transferId] = ... / s.LIQUIDITY_FEE_DENOMINATOR;
PortalFacet.sol:    if (_aavePortalFeeNumerator > s.LIQUIDITY_FEE_DENOMINATOR) ...
AssetLogic.sol:    uint256 minReceived = (_amount * _slippageTol) / s.LIQUIDITY_FEE_DENOMINATOR;
```

Recommendation: Consider creating a constant for `LIQUIDITY_FEE_DENOMINATOR`.

Connext: Solved in PR 1660.

Spearbit: Verified.

5.5.4 Access elements from storage array instead of loading them in memory

Severity: *Gas Optimization*

Context: SwapUtils.sol#L1016-L1034

Description: `SwapUtils.removeLiquidityOneToken()` function only needs the length and one element of the storage array `self.pooledTokens`. For this, the function reads the entire array in memory which costs extra gas.

```
IERC20[] memory pooledTokens = self.pooledTokens;
...
uint256 numTokens = pooledTokens.length;
...
pooledTokens[tokenIndex].safeTransfer(msg.sender, dy);
```

Recommendation: Consider using the storage array `self.pooledTokens` directly:

```
- IERC20[] memory pooledTokens = self.pooledTokens;
...
- uint256 numTokens = pooledTokens.length;
+ uint256 numTokens = self.pooledTokens.length;
...
- pooledTokens[tokenIndex].safeTransfer(msg.sender, dy);
+ self.pooledTokens[tokenIndex].safeTransfer(msg.sender, dy);
```

Connext: Solved in PR 1600.

Spearbit: Verified.

5.5.5 Send information through calldata instead of having callee query `Executor`

Severity: *Gas Optimization*

Context: Executor.sol#L35-L60, Executor.sol#L201-L211

Description: The contract (henceforth referred to as callee) called by `Executor.sol` should check `Executor.originSender()`, `Executor.origin()`, and `Executor.amount()` to permission crosschain calls. This costs extra gas because of `staticcall`'s made to an external contract.

Recommendation: Pass `originSender`, `origin`, and `amount` as part of the calldata to the callee to save the three external calls. Reading from calldata is cheaper instead.

Connext: Solved in PR 1648.

Spearbit: Verified.

5.6 Informational

5.6.1 AAVE portal debt might not be repaid in full if debt is converted to interest paying

Severity: *Informational*

Context: BridgeFacet.sol#L599-L608, BridgeFacet.sol#L723-L748, NomadFacet.sol#L146-L150, NomadFacet.sol#L176-L256

Description: The Aave portal mechanism gives routers access to a limited amount of unbacked debt which is to be used when fronting liquidity for cross-chain transfers.

The process for receiving unbacked debt is as follows:

- During message execution, the protocol checks if a single liquidity provider has bid on a liquidity auction which is handled by the relayer network.
- If the provider has insufficient liquidity, the protocol attempts to utilize AAVE unbacked debt by minting uncollateralised `aTokens` and withdrawing them from the pool. The withdrawn amount is immediately used to pay out the recipient of the bridge transfer.
- Currently the debt is fixed fee, see arc-whitelist-connext-for-v3-portals, however this might be changed in the future out of band.
- Incase this would be changed: upon repayment, AAVE will actually expect `unbackedDebt + fee + aToken interest`. The current implementation will only track `unbackedDebt + fee`, hence, the protocol will accrue bad debt in the form of interest. Eventually, the extent of this bad debt will reach a point where the `unbackedMintCap` has been reached and noone is able to pay off this debt.

I consider this to be a long-term issue that could be handled in a future upgrade, however, it is important to highlight and address these issues early.

Recommendation: Ensure this is documented and potentially add a function to allow anyone to pay off out-of-band Aave debt. It may also make sense to use part of the fee paid out to the protocol and LPs to pay off `aToken`

interest. However, the more equitable approach would be to expect routers to pay off their own interest. This serves as an incentive to pay off unbacked debt as soon as possible.

Connext: As a note, we plan on upgrading the portal implementation to make it more amenable to these types of issues in future versions outlined here

Spearbit: Acknowledged.

5.6.2 Routers pay the slippage cost for users when using AAVE credit

Severity: *Informational*

Context: BridgeFacet.sol#L723-L748

Description: When routers do the fast transfer with AAVE's credit users get `s.aavePortalFeeNumerator * _fastTransferAmount /s.LIQUIDITY_FEE_DENOMINATOR` of adopted token and `_fastTransferAmount = _args.amount * s.LIQUIDITY_FEE_NUMERATOR / s.LIQUIDITY_FEE_DENOMINATOR`. The routers get reimbursed `_args.amount` of local tokens afterward. Thus, the routers lose money if the slippage of swapping between local tokens and adopted tokens are larger than the `liquidityFee`.

```
  function _executePortalTransfer(
    bytes32 _transferId,
    bytes32 _canonicalId,
    uint256 _fastTransferAmount,
    address _router
  ) internal returns (uint256, address) {
    // Calculate local to adopted swap output if needed
    address adopted = s.canonicalToAdopted[_canonicalId];

    IAavePool(s.aavePool).mintUnbacked(adopted, _fastTransferAmount, address(this), AAVE_REFERRAL_CODE);

    // Improvement: Instead of withdrawing to address(this), withdraw directly to the user or executor
↪   to save 1 transfer
    uint256 amountWithdrawn = IAavePool(s.aavePool).withdraw(adopted, _fastTransferAmount,
↪   address(this));

    if (amountWithdrawn < _fastTransferAmount) revert
↪   BridgeFacet__executePortalTransfer_insufficientAmountWithdrawn();

    // Store principle debt
    s.portalDebt[_transferId] = _fastTransferAmount;

    // Store fee debt
    s.portalFeeDebt[_transferId] = (s.aavePortalFeeNumerator * _fastTransferAmount) /
↪   s.LIQUIDITY_FEE_DENOMINATOR;

    emit AavePortalMintUnbacked(_transferId, _router, adopted, _fastTransferAmount);

    return (_fastTransferAmount, adopted);
  }
```

Recommendation: Routers should monitor local tokens' peg and stop using AAVE's liquidity when the price is off.

Connext: Yes this is true -- and they can engage in the monitoring offchain. There is also an option for them to back the loan with the adopted asset directly, so they can more fine-tune their impact due to slippage.

Spearbit: Acknowledged.

5.6.3 Optimize max checks in `initializeSwap()`

Severity: *Informational*

Context: SwapAdminFacet.sol#L107-L175

Description: The function `initializeSwap()` `reverts` if a value is `>= ...MAX...`. Probably should revert when `> ...MAX...`.

```
function initializeSwap(...) ... {
    ...
    // Check _a, _fee, _adminFee, _withdrawFee parameters
    if (_a >= AmplificationUtils.MAX_A) revert SwapAdminFacet__initializeSwap_aExceedMax();
    if (_fee >= SwapUtils.MAX_SWAP_FEE) revert SwapAdminFacet__initializeSwap_feeExceedMax();
    if (_adminFee >= SwapUtils.MAX_ADMIN_FEE) revert SwapAdminFacet__initializeSwap_adminFeeExceedMax();
    ...
}
```

Recommendation: Change >= to >.

Connext: The values are set to specific whole numbers within SwapUtils, so code can stay as is.

Spearbit: Acknowledged.

5.6.4 All routers share the same AAVE debt

Severity: *Informational*

Context: BridgeFacet.sol#L723-L748

Description: The `mintUnbacked` amount is allocated to the calling contract (eg the *Connext Diamond that has the `BRIDGE` role permission*). Thus it is not separated to different routers, if one router does not payback its debt (in time) and has the max debt then this facility cannot be used any more.

```
function _executePortalTransfer( ... ) ... {
    ...
    IAavePool(s.aavePool).mintUnbacked(adopted, _fastTransferAmount, address(this), AAVE_REFERRAL_CODE);
    ...
}
```

Recommendation: Consider having a separate authorized contract per router, which has the right to borrow from AAVE.

Connext: A future improvement for the protocol will be around the experience of being an LP and how the contract custodies funds. One of the improvements would be thinking of funds as having an internal and external source, and making that external source more modularized. This way AAVE could be one of various different external liquidity sources. How we are planning on handling this in production is only having one router who is registered for portals. That way the concerns around portals are constrained to a single router while we develop a better, more generalized solution. Acknowledge this as an issue, and it will be addressed onchain in future upgrades and by offchain policy until then.

Spearbit: Acknowledged.

5.6.5 Careful with fee on transfer tokens on AAVE loans

Severity: *Informational*

Context: BridgeLogic.sol#L110-L140, PortalFacet.sol#L179-L197

Description: The Aave function `backUnbacked()` does not account for fee on transfer tokens. If these happen to be used then the accounting might not be right.

```
function _backLoan(...) ... {
    ...
    // back loan
    IAavePool(s.aavePool).backUnbacked(_asset, _backing, _fee);
    ...
}
```

```
library BridgeLogic {
    function executeBackUnbacked(... ) ... {
        ...
        reserve.unbacked -= backingAmount.toUint128();
        reserve.updateInterestRates(reserveCache, asset, added, 0);
        IERC20(asset).safeTransferFrom(msg.sender, reserveCache.aTokenAddress, added);
        ...
    }
}
```

Recommendation: Be careful with fee on transfer tokens with Aave loans.

Connext: I'm not aware of any tokens that have fees on one domain, and not on another, but we will make sure this is tracked against assets we add to the whitelist.

Note: We removed support for fee on transfer tokens.

Spearbit: Acknowledged.

5.6.6 Let `getTokenPrice()` also return the source of the price info

Severity: *Informational*

Context: ConnextPriceOracle.sol#L88-L107

Description: The function `getTokenPrice()` can get its prices information from multiple sources. For the caller it might be important to know which source was used.

```
function getTokenPrice(address _tokenAddress) public view override returns (uint256) {  }
```

Recommendation: Consider returning an extra value which indicates the source. For example: direct, chainlink-oracle, dex-spot, v1PriceOracle, NA

Connext: Solved in PR 1658.

Spearbit: Verified.

5.6.7 Typos in the comments of `_swapAsset()` and `_swapAssetOut()`

Severity: *Informational*

Context: AssetLogic.sol#L252-L362

Description: There are typos in the comments of `_swapAsset()` and `_swapAssetOut()`:

```
  * @notice Swaps assetIn t assetOut using the stored stable swap or internal swap pool
  function _swapAsset(... ) ...

  * @notice Swaps assetIn t assetOut using the stored stable swap or internal swap pool
  function _swapAssetOut(...) ...
```

Recommendation: Update the comments:

```
-@notice Swaps assetIn t assetOut
+@notice Swaps assetIn to assetOut
```

Connext: Solved in PR 1653.

Spearbit: Verified.

5.6.8 Consistently delete array entries in PromiseRouter

Severity: *Informational*

Context: PromiseRouter.sol#L226-L258

Description: In function `process()` of `PromiseRouter.sol` two different ways are used to clear a value: one with `delete` and the other with `= 0`. Although technically the same it better to use the same method to maintain consistency.

```
function process(bytes32 transferId, bytes calldata _message) public nonReentrant {
    ...
    // remove message
    delete messageHashes[transferId];
    // remove callback fees
    callbackFees[transferId] = 0;
    ...
  }
```

Recommendation: Consider changing the code to:

```
-callbackFees[transferId] = 0;
+delete callbackFees[transferId];
```

Connext: Solved in PR 1652.

Spearbit: Verified.

5.6.9 `getTokenPrice()` will revert if `setDirectPrice()` is set in the future

Severity: *Low Risk*

Context: ConnextPriceOracle.sol#L195-L214, ConnextPriceOracle.sol#L88-L107

Description: The `setDirectPrice()` function allows the protocol admin to update the price up to *two* seconds in the future. This impacts the `getTokenPrice()` function as the updated value may be slightly incorrect.

Recommendation: Consider checking for this behaviour or instead prevent the admin from setting the price timestamp in the future.

Connext: Solved in PR 1646.

Spearbit: Verified.

5.6.10 Roundup in `words` not optimal

Severity: *Informational*

Context: Connext copy of TypedMemView.sol#L424-L426, TypedMemView.sol#L380-L387

Description: The function `words`, which is used in the Nomad code base, tries to do a round up. Currently it adds 1 to the `len`.

```
    /**
     * @notice          The number of memory words this memory view occupies, rounded up.
     * @param memView   The view
     * @return          uint256 - The number of memory words
     */
    function words(bytes29 memView) internal pure returns (uint256) {
        return uint256(len(memView)).add(32) / 32;
    }
```

Recommendation: The code should most likely be:

```
-return uint256(len(memView)).add(32) / 32;
+return uint256(len(memView)).add(31) / 32;
```

Connext: Solved in PR 1625. Alerted the Nomad team to the problem.

Spearbit: Verified (for the Connext copy). Acknowledged (for the alert to Nomad).

5.6.11 `callback` could have capped `returnData`

Severity: *Informational*

Context: Executor.sol#L142-L243, PromiseRouter.sol#L226-L258

Description: The function `execute()` caps the result of the call to `excessivelySafeCall` to a maximum of `MAX_COPY` bytes, making sure the result is small enough to fit in a message sent back to the originator. However, when the callback is done the originator needs to be aware that the data can be capped and this fact is not clearly documented.

```
function execute(...) ... {
    ...
    (success, returnData) = ExcessivelySafeCall.excessivelySafeCall(
      _args.to,
      gas,
      isNative ? _args.amount : 0,
      MAX_COPY,
      _args.callData
    );
}
function process(bytes32 transferId, bytes calldata _message) public nonReentrant {
    ...
      // execute callback
    ICallback(callbackAddress).callback(transferId, _msg.returnSuccess(), _msg.returnData()); //
↪   returnData is capped
    ...
}
```

Recommendation: Document that the `execute()` function can have capped `returnData`, and that the `callback()` can receive chopped off data which might be interpreted in the wrong way.

Connext: Solved in PR 1670.

Spearbit: Verified.

5.6.12 Several `external` functions are not `nonReentrant`

Severity: *Informational*

Context: BridgeFacet.sol#L380-L386, PortalFacet.sol#L80-L167, RelayerFacet.sol#L130-L153

Description: The following functions don't have `nonReentrant`, while most other external functions do have such modifier.

- `bumpTransfer` of `BridgeFacet`.
- `forceReceiveLocal` of `BridgeFacet`.
- `repayAavePortal` of `PortalFacet`.
- `repayAavePortalFor` of `PortalFacet`.
- `initiateClaim` of `RelayerFacet`.

There are many swaps in the protocol and some of them should be conducted in an aggregator (not yet implemented). A lot of the aggregators use the difference between pre-swap balance and post-swap balance. (e.g. uniswap v3 router , 1inch, etc..).

While this isn't exploitable yet, there is a chance that future updates might open up an issue to exploit.

Recommendation: Consider adding `nonReentrant` on every functions that absorbs tokens/ value. Double check all other function to see if `nonReentrant` is useful.

Connext: Solved in PR 1611.

Spearbit: Verified.

5.6.13 `NomadFacet.reconcile()` has an unused argument `canonicalDomain`

Severity: *Informational*

Context: NomadFacet.sol#L122

Description: `NomadFacet.reconcile()` has an unused argument `canonicalDomain`.

Recommendation: Consider implementing one of the following:

- Comment the argument to explicitly mark that it's not used.
- This issue will be resolved if the recommendation of issue titled *"Canonical assets should be keyed on the hash of domain and id"* is followed.

Connext: Solved in PR 1523.

Note: the PR was created to address the finalized update to the nomad `BridgeRouter`.

Spearbit: Verified.

Note: The `BridgeRouter` and its interface has changed quite a lot during and after this audit. It was out of scope for this audit but it is important to have a separate review of that code, including the interface to Connext.

5.6.14 `SwapUtils._calculateSwap()` returns two values with different precision

Severity: *Informational*

Context: SwapUtils.sol#L537-L538

Description: `SwapUtils._calculateSwap()` returns `(uint256 dy, uint256 dyFee)`. `dy` is the amount of tokens a user will get from a swap and `dyFee` is the associated fee. To account for the different token decimal precision between the two tokens being swapped, a `multipliers` mapping is used to bring the precision to the same value. To return the final values, `dy` is changed back to the original token precision but `dyFee` is not.

This is an internal function and the callers adjust the fee precision back to normal, therefore severity is informational. But without documentation it is easy to miss.

Recommendation: Consider noting this difference in precision between the return values in the Natspec description of `_calculateSwap()`.

Connext: Solved in PR 1624.

Spearbit: Verified.

5.6.15 `Multicall.sol` not compatible with Natspec

Severity: *Informational*

Context: Multicall.sol#L5-L17

Description: `Multicall.sol` Natspec comment specifies:

```
/// @title Multicall - Aggregate results from multiple read-only function calls
```

However, to call those functions it uses a low level `call()` method which can call write functions as well.

```
(bool success, bytes memory ret) = calls[i].target.call(calls[i].callData);
```

Recommendation: Replace `call()` with `staticcall()` and thus preventing state changes.

Connext: Solved in PR 1612.

Spearbit: Verified.

5.6.16 reimburseRelayerFees only what is necessary

Severity: *Informational*

Context: SponsorVault.sol#L235-L271

Description: The function `reimburseRelayerFees()` gives a maximum of `relayerFeeCap` to a receiver, unless it already has a balance of `relayerFeeCap`. This implicitly means that a balance `relayerFeeCap` is sufficient. So if a receiver already has a balance only `relayerFeeCap - _to.balance` is required.

This way more recipients can be reimbursed with the same amount of funds in the SponsorVault.

```
function reimburseRelayerFees(...) ... {
    ...
    if (_to.balance > relayerFeeCap || Address.isContract(_to)) {
      // Already has fees, and the address is a contract
      return;
    }
    ...
    sponsoredFee = sponsoredFee >= relayerFeeCap ? relayerFeeCap : sponsoredFee;
    ...
  }
```

Recommendation: Consider changing the code as suggested below:

```
-sponsoredFee = sponsoredFee >= relayerFeeCap ? relayerFeeCap : sponsoredFee;
+ uint256 missingFee = relayerFeeCap - _to.balance; // already checked _to.balance <= relayerFeeCap
+sponsoredFee = sponsoredFee >= missingFee ? missingFee : sponsoredFee;
```

Connext: Solved in PR 1613.

Spearbit: Verified.

5.6.17 `safeIncreaseAllowance` and `safeDecreaseAllowance` can be replaced with `safeApprove` in `_reconcileProcessPortal`

Severity: *Informational*

Context: NomadFacet.sol#L236-L237 NomadFacet.sol#L222-L223

Description: The NomadFacet uses `safeIncreaseAllowance` after clearing the allowance. It uses `safeDecreaseAllowance` to clear the allowance. Using `safeApprove` is potentially safer in this case. Some non-standard tokens only allow the allowance to change from zero, or change to zero. Using `safeDecreaseAllowance` would potentially break the contract in a future update.

Note that `SafeApprove` has been deprecated for the concern of a front-running attack. It is only supported when setting an initial allowance or setting the allowance to zero SafeERC20.sol#L38

Recommendation: Use `safeApprove` instead.

Connext: We have decided to lean on the policy that Aave portals will not be automatically repaid.

Adding in the automatic repayment of portals adds complexity to the core codebase and leads to issues. Even with the portal repayment in place, issues such as a watcher disconnecting the `xapp` for an extended period mean we have to support out of band repayments regardless. By leaning on this as the only method of repayment, we are able to reduce the code complexity on reconcile.

Furthermore, it does not substantially reduce trust. Aave portals essentially amount to an unsecured credit line, usable by bridges. If the automatic repayment fails for any reason (i.e. due to bad pricing in the AMM), then the LP associated with the loan must be responsible for closing out the position in a trusted way.

Solved in PR 1585 by removing this code.

Spearbit: Verified.

5.6.18 Event not emitted when ERC20 and native asset is transferred together to `SponsorVault`

Severity: *Informational*

Context: SponsorVault.sol#L279-L285

Description: Any ERC20 token or native asset can be transferred to `SponsorVault` contract by calling the `deposit()` function.

It emits a `Deposit()` event logging the transferred asset and the amount. However, if the native asset and an ERC20 token are transferred in the same call only a single event corresponding to the ERC20 transfer is emitted.

Recommendation: Consider having these functions to handle ERC20 transfer and native asset transfer separately.

```
function deposit(address _token, uint256 _amount) external {
   IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);
   emit Deposit(_token, _amount, msg.sender);
}

function depositNative() external payable {
   emit Deposit(address(0), msg.value, msg.sender);
}
```

Connext: Solved in PR 1614.

Spearbit: Verified.

5.6.19 `payable` keyword can be removed

Severity: *Informational*

Context: StableSwapFacet.sol#L249, StableSwapFacet.sol#L273

Description: If a function does not need to have the native asset sent to it it is recommended to not mark it as `payable` and avoid any funds getting. `StableSwapFacet.sol` has two payable functions: `swapExact()` and `swapExactOut`, which only swap ERC20 tokens and are not expected to receive the native asset.

Recommendation: Remove `payable` keyword for `swapExact()` and `swapExactOut()`.

Connext: Solved in PR 1615.

Spearbit: Verified.

5.6.20 Improve variable naming

Severity: *Informational*

Context: BaseConnextFacet.sol#L87-L95, LibConnextStorage.sol#L244-L248, BaseConnextFacet.sol#L17, LibCrossDomainProperty.sol#L37, BridgeFacet.sol#L240-L339, BridgeFacet.sol#L644-L688

Description: Two different variables/functions with an almost identical name are prone to error.

Variable names like `_routerOwnershipRenounced` and `_assetOwnershipRenounced` do not correctly reflect their meaning as they actually refer to the ownership whitelist being renounced.

```
function _isRouterOwnershipRenounced() internal view returns (bool) {
    return LibDiamond.contractOwner() == address(0) || s._routerOwnershipRenounced;
}

/**
 * @notice Indicates if the ownership of the asset whitelist has
 * been renounced
 */
function _isAssetOwnershipRenounced() internal view returns (bool) {
...
```

```
bool _routerOwnershipRenounced;
...
// 27
bool _assetOwnershipRenounced;
```

The constant `EMPTY` is defined twice with different values. This is confusing and could lead to errors.

```
contract BaseConnextFacet {
    ...
    bytes32 internal constant EMPTY =
↪   hex"c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470";
    ...
}

library LibCrossDomainProperty {
    ...
    bytes29 public constant EMPTY = hex"ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff";
    ...
}
```

The function `xcall()` uses both `_args.transactingAssetId` and `transactingAssetId`. It is easy to mix these two, but they each have a very specific meaning and missing it introduces problems.

```
function xcall(...) ... {
    ...
    address transactingAssetId = _args.transactingAssetId == address(0)
        ? address(s.wrapper)
         : _args.transactingAssetId;
    ...
      (, uint256 amount) = AssetLogic.handleIncomingAsset(
        _args.transactingAssetId,
        ...   );
    ...
      (uint256 bridgedAmt, address bridged) = AssetLogic.swapToLocalAssetIfNeeded(
        ...,
        transactingAssetId,
        ...  );
     ...
}
```

In the `_handleExecuteTransaction` function of `BridgeFacet`, `_args.amount` and `_amount` are used. In this function:

- `_args.amount` is equal to bridged_amount;

- `_amount` is equal to bridged_amount - liquidityFee (and potentially swapped amount).

Recommendation: Rename the variables/functions to improve comprehension.

Connext: Solved in PR 1608 and PR 1629.

Spearbit: Verified.

5.6.21 `onlyRemoteRouter` can be circumvented

Severity: *Informational*

Context: Router.sol#L56-L58, BridgeRouter.sol#L112-L117, Replica.sol#L179-L204

Description: The Code4rena contest suggested a modification Code4arena-254 that was fixed in BaseConnextFacet-fix. However, the change has not been applied to Router.sol#L56-L58 which is currently in use.

The modifier `onlyRemoteRouter()` can be mislead if the sender parameter has the value `0`. The modifier uses `_m.sender()` from the received message by Nomad. Assuming all checks of Nomad work as expected this value cannot be `0` as it originates from a `msg.sender` in `Home.sol`.

```
contract Replica is Version0, NomadBase {
    function process(bytes memory _message) public returns (bool _success) {
        ...
        bytes29 _m = _message.ref(0);
        ...
        // ensure message has been proven
        bytes32 _messageHash = _m.keccak();
        require(acceptableRoot(messages[_messageHash]), "!proven");
        ...
        IMessageRecipient(_m.recipientAddress()).handle(
            _m.origin(),
            _m.nonce(),
            _m.sender(),
            _m.body().clone()
        );
        ...
    }
}
```

```
contract BridgeRouter is Version0, Router {
    function handle(uint32 _origin,uint32 _nonce,bytes32 _sender,bytes memory _message)
        external override onlyReplica onlyRemoteRouter(_origin, _sender) {
            ...
    }
}
```

```
abstract contract Router is XAppConnectionClient, IMessageRecipient {
    ...
    modifier onlyRemoteRouter(uint32 _origin, bytes32 _router) {
        require(_isRemoteRouter(_origin, _router), "!remote router");
        _;
    }
    function _isRemoteRouter(uint32 _domain, bytes32 _router) internal view returns (bool) {
        return remotes[_domain] == _router;  // if _router == 0 then this is true for random _domains
    }
}
```

Recommendation: To be extra careful, consider applying the changes also to `Router.sol`:

```
function _isRemoteRouter(uint32 _domain, bytes32 _router) internal view returns (bool) {
    return s.remotes[_domain] == _router && _router != bytes32(0);
}
```

Connext: Solved in PR 1616.

Spearbit: Verified.

5.6.22 Some dust not accounted for in `reconcile()`

Severity: *Informational*

Context: BridgeFacet.sol#L571-L628, NomadFacet.sol#L118-L165

Description: The function `_handleExecuteLiquidity()` in `BridgeFacet` takes care of rounding issues in `toSwap / pathLen`. However, the inverse function `reconcile()` in `NomadFacet()` does not do that.

So, tiny amounts of tokens (dust) are not accounted for in `reconcile()`.

```
contract BridgeFacet is BaseConnextFacet {
    ...
    function _handleExecuteLiquidity(...) ... {
       ...
       // For each router, assert they are approved, and deduct liquidity.
       uint256 routerAmount = toSwap / pathLen;
       for (uint256 i; i < pathLen - 1; ) {
           s.routerBalances[_args.routers[i]][_args.local] -= routerAmount;
           unchecked { ++i;  }
        }
       // The last router in the multipath will sweep the remaining balance to account for remainder
↪   dust.
       uint256 toSweep = routerAmount + (toSwap % pathLen);
       s.routerBalances[_args.routers[pathLen - 1]][_args.local] -= toSweep;
       }
    }
}
```

```
contract NomadFacet is BaseConnextFacet {
    ...
    function reconcile(...) ... {
        ...
        uint256 routerAmt = toDistribute / pathLen;
        for (uint256 i; i < pathLen; ) {
            s.routerBalances[routers[i]][localToken] += routerAmt;
            unchecked { ++i;  }
        }
    }
}
```

Recommendation: Consider giving the last `router` the remaining tokens (dust) in function `reconcile()`. Alternatively add a comment that some dust tokens are neglected.

Connext: Solved in PR 1617.

Spearbit: Verified.

5.6.23 Careful with the decimals of `BridgeToken`s

Severity: *Informational*

Context: BridgeRouter.sol#L226-L334, BridgeToken.sol#L93-L119, initializeSwap.ts#L109-L110 SwapAdminFacet.sol#L107-L175

Description: The `BridgeRouter` sends token details including the `decimals()` over the nomad bridge to configure a new deployed token. After setting the hash with `setDetailsHash()` anyone can call `setDetails()` on the token to set the details.

The `decimals()` are mainly used for user interfaces so it might not be a large problem when the `setDetails()` is executed at later point in time. However `initializeSwap()` also uses `decimals()`, this is called via offchain code. In the example code of `initializeSwap.ts` it retrieves the decimals() from the deployed token on the `destination` chain. This introduces a race condition between `setDetails()` and `initializeSwap.ts`, depending on which is executed first, the swaps will have different results.

Note: It could also break the `ConnextPriceOracle`

```
contract BridgeRouter is Version0, Router {
    ...
    function _send( ... ) ... {
        ...
        if (tokenRegistry.isLocalOrigin(_token)) {
            ...
            // query token contract for details and calculate detailsHash
            _detailsHash = BridgeMessage.getDetailsHash(_t.name(), _t.symbol(), _t.decimals());
        } else {
            ...
        }
    }
    function _handleTransfer(...) ... {
        ...
        if (tokenRegistry.isLocalOrigin(_token)) {
            ...
        } else {
            ...
            IBridgeToken(_token).setDetailsHash(_action.detailsHash()); // so hash is set now
        }
    }
}
```

```
contract BridgeToken is Version0, IBridgeToken, OwnableUpgradeable, ERC20 {
    ...
    function setDetails(..., uint8 _newDecimals) ... {  // can be called by anyone
        ...
        require(
            _isFirstDetails || BridgeMessage.getDetailsHash(..., _newDecimals) == detailsHash,
            "!committed details"
        );
        ...
        token.decimals = _newDecimals;
        ...
        }
    }
```

Example script: `initializeSwap.ts`

```
  const decimals = await Promise.all([
        (await ethers.getContractAt("TestERC20", local)).decimals(),
        (await ethers.getContractAt("TestERC20", adopted)).decimals(), // setDetails might not have
        ↪  been done
      ]);

      const tx = await connext.initializeSwap(...,  decimals, ... );
  );
```

```
contract SwapAdminFacet is BaseConnextFacet {
    ...
    function initializeSwap(..., uint8[] memory decimals, ... ) ... {
        ...
        for (uint8 i; i < numPooledTokens; ) {
            ...
            precisionMultipliers[i] = 10**uint256(SwapUtils.POOL_PRECISION_DECIMALS - decimals[i]);
            ...
       }
    }
}
```

Recommendation: Set the decimals of the deployed token on the destination chain in a deterministic way. Or use the token decimals on the origination chain when calling `initializeSwap()`, and adapt example code to prevent mistakes.

Connext: Will be solved in deployment scripts. This PR adds a comment to the test deployment scripts: PR 1627.

Spearbit: Acknowledged.

5.6.24 Incorrect comment about ERC20 approval to zero-address

Severity: *Informational*

Context: AssetLogic.sol#L289-L290

Description: The linked code notes in a comment:

```
// NOTE: if pool is not registered here, then the approval will fail
// as it will approve to the zero-address
SafeERC20.safeIncreaseAllowance(IERC20(_assetIn), address(pool), _amount);
```

This is not always true. The ERC20 spec doesn't have this restriction and ERC20 tokens based on solmate also don't revert on approving to zero-address.

There is no risk here as the following line of code for zero-address pools will revert.

```
return (pool.swapExact(_amount, _assetIn, _assetOut, minReceived), _assetOut);
```

Recommendation: Update the comments.

Connext: Solved in PR 1618.

Spearbit: Verified.

5.6.25 Native asset is delivered even if the wrapped asset is transferred

Severity: *Informational*

Context: BridgeFacet.sol#L292-L293, AssetLogic.sol#L75-L80, AssetLogic.sol#L140-L145

Description: Connext delivers the native asset on the destination chain even if the wrapped asset was transferred. This is because on the source chain the native asset is converted to the wrapped asset, and then the distinction is lost.

On the destination chain it is not possible to know which of these two assets was transferred, and hence a choice is made to transfer the native asset.

```
if (_assetId == address(0)) revert AssetLogic__transferAssetFromContract_notNative();

if (_assetId == address(s.wrapper)) {
  // If dealing with wrapped assets, make sure they are properly unwrapped
  // before sending from contract
  s.wrapper.withdraw(_amount);
  Address.sendValue(payable(_to), _amount);
} else {
...
```

Recommendation: Consider removing the capability of transferring native asset through Connext. Users can transfer wrapped assets, and the wrapping and unwrapping can happen outside the Connext system. This simplifies the code by removing a few branches to handle native assets and wrapped assets differently from the usual ERC20 tokens.

Connext: Removed native asset handling in PR 1641.

Spearbit: Verified.

5.6.26 Entire transfer amount is borrowed from AAVE Portal when a router has insufficient balance

Severity: *Informational*

Context: BridgeFacet.sol#L601-L608

Description: If the router picked by the Sequencer doesn't have enough balance to transfer the required amount, it can borrow the entire amount from Aave Portal. For a huge amount, it will block borrowing for other routers since there is a limit on the total maximum amount that can be borrowed.

Recommendation: Borrow the difference between the transfer amount and router's balance.

Connext: I think keeping the original code is likely best, closing the PR!

Spearbit: Acknowledged (the code would indeed get far more complicated trying to solve this).

5.6.27 Unused variable

Severity: *Informational*

Context: BridgeFacet.sol#L265

Description: The variable `message` is not used after declaration.

```
bytes memory message;
```

Recommendation: Remove this variable declaration.

Connext: Solved in PR 1523.

Spearbit: Verified.

5.6.28 Incorrect Natspec for adopted and canonical asset mappings

Severity: *Informational*

Context: LibConnextStorage.sol#L172-L184

Description: `adoptedToCanonical` maps adopted assets to canonical assets, but is described as a "Mapping of canonical to adopted assets"; `canonicalToAdopted` maps canonical assets to adopted assets, but is described as a "Mapping of adopted to canonical assets".

```
// /**
// * @notice Mapping of canonical to adopted assets on this domain
// * @dev If the adopted asset is the native asset, the keyed address will
// * be the wrapped asset address
// */
// 12
mapping(address => TokenId) adoptedToCanonical;
// /**
// * @notice Mapping of adopted to canonical on this domain
// * @dev If the adopted asset is the native asset, the stored address will be the
// * wrapped asset address
// */
// 13
mapping(bytes32 => address) canonicalToAdopted;
```

Recommendation: Update the Natspec comment to correctly describe the variables.

Connext: Solved in PR 1588.

Spearbit: Verified.

5.6.29 Use of SafeMath for solc >= 0.8

Severity: *Informational*

Context: AmplificationUtils.sol#L5-L16, SwapUtils.sol#L4-L21, ConnextPriceOracle.sol#L45, GovernanceRouter.sol#L20

Description: `AmplificationUtils`, `SwapUtils`, `ConnextPriceOracle`, `GovernanceRouter.sol` use `SafeMath`. Since `0.8.0`, arithmetic in `solidity` reverts if it overflows or underflows, hence there is no need to use openzeppelin's `SafeMath` library.

Recommendation: Remove SafeMath as a dependency and use vanilla arithmetic operators.

Connext: Solved in PR 1619. `GovernanceRouter.sol` is a Nomad contract, so will be handled by Nomad team.

Spearbit: Verified and acknowledged.

6 Appendix: Contract architecture overview