Connext Amarok contest Findings & Analysis Report

2022-10-17

TABLE OF CONTENTS

Overview

ABOUT C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Connext Amarok smart contract system written in Solidity. The audit contest took place between June 8—June 19 2022.

WARDENS

81 Wardens contributed reports to the Connext Amarok contest:

xiaoming90

unforgiven

WatchPug (jtp and ming)

csanuragjain

0x1f8b

Chom

hyh

Ruhum

0xmint

cloudjunky

shenwilly

codexploder

BowTiedWardens (BowTiedHeron, BowTiedPickle, m4rio_eth, Dravee, and BowTiedFirefox)

Czar102

0x52

IllIIlI

hansfriese

SmartSek (0xDjango and hake)

cccz

slywaters

0xKitsune

0xNineDec

Lambda

0xkatana

joestakey

defsec

MiloTruck

cryptphi

0xf15ers (remora and twojoy)

oyc_109

0x29A (0x4non and rotcivegaf)

minhquanym

GimelSec (rayn and sces60107)

simon135

robee

Waze

TomJ

catchup

_Adam

c3phas

ElKu

Kaiziron

0xNazgul

asutorufos

k

fatherOfBlocks

sach1r0

Funen

bardamu

cmichel

JMukesh

sorrynotsorry

Jujic

kenta

TerrierLover

ch13fd357r0y3r

SooYa

tintin

auditor0517

jayjonah8

obtarian

zzzitron

Metatron

Tomio

UnusualTurtle

apostle0x01

kaden

rfa

Fitraldys

ignacio

nahnah

Randyyy

This contest was judged by 0xleastwood.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 26 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 20 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 57 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 44 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Connext Amarok contest repository, and is composed of 24 smart contracts (and 10 libraries) written in the Solidity programming language and includes 3765 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (6)

[H-01] PORTCALFACET.REPAYAAVEPORTAL() CAN TRIGGER AN UNDERFLOW OF ROUTERBALANCES

Submitted by Ruhum, also found by 0x1f8b and WatchPug

PortalFacet.sol#L80-L113

The caller of `repayAavePortal()` can trigger an underflow to arbitrarily increase the caller's balance through an underflow.

Proof of Concept

```
// Relevant code sections:

// PortalFacet.sol
  function repayAavePortal(
    address _local,
    uint256 _backingAmount,
    uint256 _feeAmount,
    uint256 _maxIn,
    bytes32 _transferId
  ) external {
    uint256 totalAmount = _backingAmount + _feeAmount; // in adopted
    uint256 routerBalance = s.routerBalances[msg.sender][_local]; // in local

    // Sanity check: has that much to spend
    if (routerBalance < _maxIn) revert PortalFacet__repayAavePortal_insufficientFunds();

    // Need to swap into adopted asset or asset that was backing the loan
    // The router will always be holding collateral in the local asset while the loaned as
    // is the adopted asset

    // Swap for exact `totalRepayAmount` of adopted asset to repay aave
    (bool success, uint256 amountIn, address adopted) = AssetLogic.swapFromLocalAssetIfNee
      _local,
      totalAmount,
      _maxIn
    );

    if (!success) revert PortalFacet__repayAavePortal_swapFailed();

    // decrement router balances
    unchecked {
      s.routerBalances[msg.sender][_local] -= amountIn;
    }

    // back loan
    _backLoan(_local, _backingAmount, _feeAmount, _transferId);
  }

// AssetLogic.sol
  function swapFromLocalAssetIfNeededForExactOut(
    address _asset,
    uint256 _amount,
    uint256 _maxIn
  )
    internal
    returns (
      bool,
      uint256,
      address
    )
  {
    AppStorage storage s = LibConnextStorage.connextStorage();

    // Get the token id
    (, bytes32 id) = s.tokenRegistry.getTokenId(_asset);

    // If the adopted asset is the local asset, no need to swap
    address adopted = s.canonicalToAdopted[id];
    if (adopted == _asset) {
      return (true, _amount, _asset);
    }

    return _swapAssetOut(id, _asset, adopted, _amount, _maxIn);
  }
```

First, call `repayAavePortal()` where `_backingAmount + _feeAmount > s.routerBalances[msg.sender][_local] && _maxIn > s.routerBalances[msg.sender][_local]`. That will trigger the call to the AssetLogic contract:

```
    (bool success, uint256 amountIn, address adopted) = AssetLogic.swapFromLocalAssetIfNee
      _local,
      totalAmount,
      _maxIn
    );
```

By setting `_local` to the same value as the adopted asset, you trigger the following edge case:

```
    address adopted = s.canonicalToAdopted[id];
    if (adopted == _asset) {
      return (true, _amount, _asset);
    }
```

So the `amountIn` value returned by `swapFromLocalAssetIfNeededForExactOut()` is the `totalAmount` value that was passed to it. And `totalAmount == _backingAmount + _feeAmount`.

Meaning the `amountIn` value is user-specified for this edge case. Finally, we reach the following line:

```
    unchecked {
      s.routerBalances[msg.sender][_local] -= amountIn;
    }
```

`amountIn` (user-specified) is subtracted from the `routerBalances` in an `unchecked` block. Thus, the attacker is able to trigger an underflow and increase their balance arbitrarily high. The `repayAavePortal()` function only verifies that `routerBalance < _maxIn`.

Here's a test as PoC:

```
// PortalFacet.t.sol

  function test_PortalFacet_underflow() public {
    s.routerPermissionInfo.approvedForPortalRouters[router] = true;

    uint backing = 2 ether;
    uint fee = 10000;
    uint init = 1 ether;

    s.routerBalances[router][_local] = init;
    s.portalDebt[_id] = backing;
    s.portalFeeDebt[_id] = fee;

    vm.mockCall(s.aavePool, abi.encodeWithSelector(IAavePool.backUnbacked.selector), abi.e
    vm.prank(router);
    this.repayAavePortal(_local, backing, fee, init - 0.5 ether, _id);

    // balance > init => underflow
    require(s.routerBalances[router][_local] > init);
  }
```

Recommended Mitigation Steps

After the call to `swapFromLocalAssetIfNeededForExactOut()` you should add the following check:

```
if (_local == adopted) {
  require(routerBalance >= amountIn);
}
```

LayneHaber (Connext) confirmed and resolved:

> connext/nxtp@ac95c1b

0xleastwood (judge) commented:

> This is entirely valid and a really severe issue. If the local asset is the adopted asset, `AssetLogic.swapFromLocalAssetIfNeededForExactOut()` will return `amountIn == totalAmount`. So in order to overflow `routerBalances`, the router just needs to provide `_backingAmount + _feeAmount` inputs that sum to exceed the router's current balance.

[H-02] WRONG IMPLEMENTATION OF WITHDRAWADMINFEES() CAN CAUSE

THE ADMINFEES TO BE CHARGED MULTIPLE TIMES AND THEREFORE CAUSE USERS’ FUND LOSS

Submitted by WatchPug

SwapUtils.sol#L1053-L1062

```
function withdrawAdminFees(Swap storage self, address to) internal {
  IERC20[] memory pooledTokens = self.pooledTokens;
  for (uint256 i = 0; i < pooledTokens.length; i++) {
    IERC20 token = pooledTokens[i];
    uint256 balance = self.adminFees[i];
    if (balance != 0) {
      token.safeTransfer(to, balance);
    }
  }
}
```

`self.adminFees[i]` should be reset to 0 every time it’s withdrawn. Otherwise, the `adminFees` can be withdrawn multiple times.

The admin may just be unaware of this issue and casualty `withdrawAdminFees()` from time to time, and rug all the users slowly.

Recommended Mitigation Steps

Change to:

```
function withdrawAdminFees(Swap storage self, address to) internal {
  IERC20[] memory pooledTokens = self.pooledTokens;
  for (uint256 i = 0; i < pooledTokens.length; i++) {
    IERC20 token = pooledTokens[i];
    uint256 balance = self.adminFees[i];
    if (balance != 0) {
      self.adminFees[i] = 0;
      token.safeTransfer(to, balance);
    }
  }
}
```

LayneHaber (Connext) confirmed and resolved:

> connext/nxtp@8eef974

0xleastwood (judge) commented:

> Completely agree with the validity of this finding. Even if the admin was *not* malicious, the bug will still continue to withdraw additional fees which were not included as part of the swap calculations. LPs would lose considerable value as a result.

[H-03] ROUTER OWNER COULD STEAL ALL THE FUNDS WITHIN SPONSORVAULT

Submitted by xiaoming90

BridgeFacet.sol#L541
SponsorVault.sol#L196

Assume the following:

For simplicity sake, only two (2) routers exist within Connext. Gas, relayer, callback fees and slippage are ignored.

An attacker owns Router A. Router A has `1,000,000` oUSDC on Optimism Domain/Chain

Router B has only `100` oUSDC on Optimism Domain/Chain

The liquidity fee is 5% for fast transfer service

SponserVault will reimbursed 50% of the liquidity fee incurred by the users

At this point, attacker balances are as follows (2,000,000 USDC in total)

Attacker's wallet in Ethereum = `1,000,000` USDC

Attacker's wallet in Optimism = `0` oUSDC

Attacker's router in Optimism = `1,000,000` oUSDC

First, the attacker attempts to transfer an extremely large amount - `1,000,000` USDC from attacker's address in Ethereum to attacker's address in Optimism Chain. The transfer amount should be much larger than the rest of the router's liquidity so that only attacker's router is capable of providing the liqudity.

In our example, since Router B does not have sufficient liquidity to faciliate the fast transfer, Router B will not be selected by the Sequencer. Since only Router A has sufficient liquidity, Router A, which is owned by the attacker, will faciliate the fast transfer and selected by the Sequencer.

Since the liquidity fee is 5%, Router A only need to supply `950,000` oUSDC on `execute`. The Sponsor will then reimburse 50% of the liquidity fee, which is`25,000` oUSDC in total. The final amount of oUSDC send to the attacker's wallet address in Optimism will be `975,000` oUSDC.

At this point, attacker balances are as follows (1,025,000 USDC in total)

Attacker's wallet in Ethereum = `0` USDC

Attacker's wallet in Optimism = `975,000` oUSDC

Attacker's router in Optimism = `50,000` oUSDC

When the nomad message arrives, the attacker will be reimbursed `1,000,000` oUSDC when `BridgeFacet._reconcile` is triggered.

At this point, attacker balances are as follows (2,025,000 USDC in total)

Attacker's wallet in Ethereum = `0` USDC

> Attacker's wallet in Optimism = `975,000` oUSDC
>
> Attacker's router in Optimism = `50,000` + `1,000,000` oUSDC

Attacker earned `25,000` USDC, and SponsorVault lost `25,000` USDC.

Impact

Router owner can intentionally perform many large transfer between their own wallets in two different domain to siphon all the funds from the SponsorVault, and then proceed to withdraw all liquidity from his router.

Recommended Mitigation Steps

Although having a sponsor to subside the liqudity fee to encourage users to use sponsor's chain, this subsidy can be gamed by malicious actors for their own benefits. It is recommended to reconsider the need of having a sponsor in Connext as extreme care have to be taken in its design to ensure that it will not be exploited.

LayneHaber (Connext) acknowledged and commented:

> The `SponsorVault` is not mandatory for the bridge flow, and the entire point of the vault option is to allow domains to subsidize fees for users transferring funds there. This is incredibly useful for new domains, that have no default bridge and want to remove any friction for users to get to their chain. Sponsor vault funders should be informed there is no way to enforce only legitimate users get the funds and it is inherently vulnerable to sybil attacks. In our conversations with potential sponsors, they are aware of these issues and are still willing to fund sponsor vaults to a limited capacity.

0xleastwood (judge) commented:

> It seems that it would be easy for routers to sybil attack the protocol and continuously drain the sponsor vault of all its funds. While I understand this might not be an issue when the set of routers is trusted, however, as the protocol continues to become more decentralized, this would be a likely path of attack. I also agree with the current risk even though users' funds aren't at direct risk, the functionality of the sponsor vault is rendered useless and the router profits from this attack.

[H-04] IN EXECUTE() THE AMOUNT ROUTERS PAY IS WHAT USER SIGNED, BUT IN _RECONCILE() THE AMOUNT ROUTERS GET IS WHAT NOMAD SENDS AND THESE TWO AMOUNTS ARE NOT NECESSARY EQUAL BECAUSE OF SLIPPAGE IN ORIGINAL DOMAIN

Submitted by unforgiven

BridgeFacet.sol#L526-L616
BridgeFacet.sol#L753-L803
BridgeFacet.sol#L398-L428

BridgeFacet.sol#L345-L351

Routers pay for transaction in destination domain then nomad messages come and routers get paid again. but the amount routers pay in `execute()` are what transaction sender signed and the amount routers receive is what nomad sends and handles in `_reconcile()` but this two amount can be different because of slippage and swap that happens in `xcall()` because the amount sent in nomad message is the result of `swapToLocalAssetIfNeeded()`.
So it's possible for routers to lose funds if some slippage happens in that swap.

Proof of Concept

This is `xcall()` code:

```
function xcall(XCallArgs calldata _args) external payable whenNotPaused nonReentrant ret
  // Sanity checks.
  {
    // Correct origin domain.
    if (_args.params.originDomain != s.domain) {
      revert BridgeFacet__xcall_wrongDomain();
    }

    // Recipient is defined.
    if (_args.params.to == address(0)) {
      revert BridgeFacet__xcall_emptyTo();
    }

    // If callback address is not set, callback fee should be 0.
    if (_args.params.callback == address(0) && _args.params.callbackFee > 0) {
      revert BridgeFacet__xcall_nonZeroCallbackFeeForCallback();
    }

    // Callback is contract if supplied.
    if (_args.params.callback != address(0) && !Address.isContract(_args.params.callback
      revert BridgeFacet__xcall_callbackNotAContract();
    }
  }

  bytes32 transferId;
  bytes memory message;
  XCalledEventArgs memory eventArgs;
  {
    // Get the remote BridgeRouter address; revert if not found.
    bytes32 remote = _mustHaveRemote(_args.params.destinationDomain);

    // Get the true transacting asset ID (using wrapper instead of native, if applicable
    address transactingAssetId = _args.transactingAssetId == address(0)
      ? address(s.wrapper)
      : _args.transactingAssetId;

    // Check that the asset is supported -- can be either adopted or local.
    ConnextMessage.TokenId memory canonical = s.adoptedToCanonical[transactingAssetId];
    if (canonical.id == bytes32(0)) {
      // Here, the asset is *not* the adopted asset. The only other valid option
      // is for this asset to be the local asset (i.e. transferring madEth on optimism)
      // NOTE: it *cannot* be the canonical asset. the canonical asset is only used on
      // the canonical domain, where it is *also* the adopted asset.
      if (s.tokenRegistry.isLocalOrigin(transactingAssetId)) {
        // revert, using a token of local origin that is not registered as adopted
        revert BridgeFacet__xcall_notSupportedAsset();
      }

      (uint32 canonicalDomain, bytes32 canonicalId) = s.tokenRegistry.getTokenId(transac
      canonical = ConnextMessage.TokenId(canonicalDomain, canonicalId);
    }

    transferId = _getTransferId(_args, canonical);
    s.nonce += 1;

    // Store the relayer fee
    s.relayerFees[transferId] = _args.params.relayerFee;

    // Transfer funds of transacting asset to the contract from the user.
    // NOTE: Will wrap any native asset transferred to wrapped-native automatically.
    (, uint256 amount) = AssetLogic.handleIncomingAsset(
      _args.transactingAssetId,
      _args.amount,
      _args.params.relayerFee + _args.params.callbackFee
    );

    // Swap to the local asset from adopted if applicable.
    (uint256 bridgedAmt, address bridged) = AssetLogic.swapToLocalAssetIfNeeded(
      canonical,
      transactingAssetId,
      amount,
      _args.params.slippageTol
    );

    // Transfer callback fee to PromiseRouter if set
    if (_args.params.callbackFee != 0) {
      s.promiseRouter.initCallbackFee{value: _args.params.callbackFee}(transferId);
    }

    message = _formatMessage(_args, bridged, transferId, bridgedAmt);
    s.xAppConnectionManager.home().dispatch(_args.params.destinationDomain, remote, mess

    // Format arguments for XCalled event that will be emitted below.
    eventArgs = XCalledEventArgs({
      transactingAssetId: transactingAssetId,
      amount: amount,
      bridgedAmt: bridgedAmt,
      bridged: bridged
    });
  }

  // emit event
  emit XCalled(transferId, _args, eventArgs, s.nonce - 1, message, msg.sender);

  return transferId;
}
```

As you can see it swaps what user sent to `LoccalAsset` which the amount is `bridgedAmt` and then send value of `bridgedAmt` to nomad bridge `message = _formatMessage(_args, bridged, transferId, bridgedAmt)`.
But the amount user signed in `_args.amount` is different and that what user sends to contract.
The reasons that `bridgedAmt` could be different than `_args.amount` is:
1- deflationary tokens in transferring from user.
2- slippage in swap to local token.
This is `_reconcile()` code:

```
function _reconcile(uint32 _origin, bytes memory _message) internal {
  // Parse tokenId and action from the message.
  bytes29 msg_ = _message.ref(0).mustBeMessage();
  bytes29 tokenId = msg_.tokenId();
  bytes29 action = msg_.action();
```

```
// Assert that the action is valid.
if (!action.isTransfer()) {
  revert BridgeFacet__reconcile_invalidAction();
}

// Load the transferId.
bytes32 transferId = action.transferId();

// Ensure the transaction has not already been handled (i.e. previously reconciled).
if (s.reconciledTransfers[transferId]) {
  revert BridgeFacet__reconcile_alreadyReconciled();
}

// NOTE: `tokenId` and `amount` must be in plaintext in the message so funds can *only
// `handle`. They are both used in the generation of the `transferId` so routers must
// correctly to be reimbursed.

// Get the appropriate local token contract for the given tokenId on this chain.
// NOTE: If the token is of remote origin and there is no existing representation toke
// the TokenRegistry will deploy a new one.
address token = s.tokenRegistry.ensureLocalToken(tokenId.domain(), tokenId.id());

// Load amount once.
uint256 amount = action.amnt();

// Mint tokens if the asset is of remote origin (i.e. is representational).
// NOTE: If the asset IS of local origin (meaning it's canonical), then the tokens wil
// in escrow in this contract (from previous `xcall`s).
if (!s.tokenRegistry.isLocalOrigin(token)) {
  IBridgeToken(token).mint(address(this), amount);

  // Update the recorded `detailsHash` for the token (name, symbol, decimals).
  // TODO: do we need to keep this
  bytes32 details = action.detailsHash();
  IBridgeToken(token).setDetailsHash(details);
}

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

// If the transfer was executed using fast-liquidity provided by routers, then this va
// to the participating routers.
// NOTE: If the transfer was not executed using fast-liquidity, then the funds will be
// execution (i.e. funds will be delivered to the transfer's recipient in a subsequent
address[] memory routers = s.routedTransfers[transferId];

// If fast transfer was made using portal liquidity, we need to repay
// FIXME: routers can repay any-amount out-of-band using the `repayAavePortal` method
// or by interacting with the aave contracts directly
uint256 portalTransferAmount = s.portalDebt[transferId] + s.portalFeeDebt[transferId];

uint256 toDistribute = amount;
uint256 pathLen = routers.length;
if (portalTransferAmount != 0) {
  // ensure a router took on credit risk
  if (pathLen != 1) revert BridgeFacet__reconcile_noPortalRouter();
  toDistribute = _reconcileProcessPortal(amount, token, routers[0], transferId);
}

if (pathLen != 0) {
  // fast liquidity path
  // Credit each router that provided liquidity their due 'share' of the asset.
  uint256 routerAmt = toDistribute / pathLen;
  for (uint256 i; i < pathLen; ) {
    s.routerBalances[routers[i]][token] += routerAmt;
    unchecked {
      i++;
    }
  }
}
```

```
    emit Reconciled(transferId, _origin, routers, token, amount, msg.sender);
  }
```

As you can see it uses amount in message to calculate what router should receive.
This is `_handleExecuteLiquidity()` code which is used in `execute()`:

```
  function _handleExecuteLiquidity(
    bytes32 _transferId,
    bool _isFast,
    ExecuteArgs calldata _args
  ) private returns (uint256, address) {
    uint256 toSwap = _args.amount;

    // If this is a fast liquidity path, we should handle deducting from applicable router
    // If this is a slow liquidity path, the transfer must have been reconciled (if we've
    // and the funds would have been custodied in this contract. The exact custodied amour
    // (since the amount is hashed in the transfer ID itself) - thus, no updates are requi
    if (_isFast) {
      uint256 pathLen = _args.routers.length;

      // Calculate amount that routers will provide with the fast-liquidity fee deducted.
      toSwap = _getFastTransferAmount(_args.amount, s.LIQUIDITY_FEE_NUMERATOR, s.LIQUIDITY

      // Save the addressess of all routers providing liquidity for this transfer.
      s.routedTransfers[_transferId] = _args.routers;

      // If router does not have enough liquidity, try to use Aave Portals.
      // only one router should be responsible for taking on this credit risk, and it shou
      // deal with transfers expecting adopted assets (to avoid introducing runtime slippa
      if (
        !_args.params.receiveLocal &&
        pathLen == 1 &&
        s.routerBalances[_args.routers[0]][_args.local] < toSwap &&
        s.aavePool != address(0)
      ) {
        if (!s.routerPermissionInfo.approvedForPortalRouters[_args.routers[0]])
          revert BridgeFacet__execute_notApprovedForPortals();

        // Portal provides the adopted asset so we early return here
        return _executePortalTransfer(_transferId, toSwap, _args.local, _args.routers[0]);
      } else {
        // for each router, assert they are approved, and deduct liquidity
        uint256 routerAmount = toSwap / pathLen;
        for (uint256 i; i < pathLen; ) {
          // decrement routers liquidity
          s.routerBalances[_args.routers[i]][_args.local] -= routerAmount;

          unchecked {
            i++;
          }
        }
      }
    }
```

As you can see it uses the amount defined in `ExecuteArgs` to see how much routers should pay.

Because of these two issues (deflationary tokens and swap slippage) attacker could fool protocol to spend more than what he transferred to protocol. This could be seen as two bugs.

Tools Used

VIM

Recommended Mitigation Steps

Update spending amount based on (deflationary tokens and swap slippage).

LayneHaber (Connext) disputed and commented:

> I think there is a misunderstanding here — the user takes on the slippage risk both into and out of the local assets, and the router has consistent returns on what was bridged.
>
> On `xcall`, the user swaps the amount put in for the local asset. This incurs some slippage, and only the amount of the local asset is bridged directly. It is the bridged amount that the router should supply liquidity for, and take fees on. Once the router supplies liquidity in `execute` (bridged amount minus the fees), then it is swapped for the local asset and sent to the user. The user may get some different amount here, but it is the user who is impacted by this slippage. On handle, the router is credited the bridged amount.
>
> However, there was a separate bug where the `transferId` was generated with the wrong `amount` on execute, so that could be where the confusion is coming from.

0xleastwood (judge) commented:

> I actually agree with the warden here, it seems that they're right about the issue but they just failed to mention the main reason why its an issue is because `transferId` is calculated using `_args.amount` which does not necessarily equal `bridgedAmt` due to slippage. Therefore, routers may end up fronting too much liquidity and receive considerably less when the bridge transfer is eventually reconciled. This seems rather severe as the user will receive the full transfer amount without slippage. This could be abused to drain routers on low liquidity tokens.

LayneHaber (Connext) commented:

> Right — I agree that the problems outlined here would be the true consequences for a mismatched `transferId`. If the question is to take the action outlined here — specifically to keep this open and downgrade #227 as a QA — that would work with me.

LayneHaber (Connext) resolved:

> connext/nxtp@f41a156

[H-05] ROUTERS ARE NOT ENFORCED TO REPAY AAVE PORTAL LOAN

Submitted by xiaoming90

BridgeFacet.sol#L984

Background

AAVE Portal

AAVE portal provides a trusted credit line that allows bridges to take on an unbacked position, and Connext intents to use this credit line to provide fast-liquidity for its users in the event the routers do not have sufficient liquidity.

Connext will assign one (1) router to be responsible for taking on credit risk of borrowing an unbacked position from AAVE portal as per Source Code.

Under normal circumstance, the `BridgeFacet._reconcile` function will automatically repay back the loan to AAVE portal when the nomad message arrives. However, if the repayment fails for certain reason, Connext expects that the router will use the `repayAavePortal` function out-of-band to help Connext to repay the loan.

Ultimately, it is Connext that take on the credit risk because AAVE portal only provides a trusted credit line to Connext, but not to the individual routers.

Nomad Message

When nomad message arrives, it will call `BridgeFacet.handle` function, which will in turn trigger the internal `_reconcile` function. Note that the `handle` or `_reconcile` function cannot be reverted under any circumstances because nomad message cannot be reprocessed on the nomad side.

Proof-of-Concept

Alice transfers `1,000,000` DAI from Ethereum domain to Polygon domain

None of the existing routers have sufficient liquidity, thus the sequencer decided that AAVE Portal should be used

Bob's router has been selected to take on the credit risk for the unbacked position, and Connext proceeds to borrow `1,000,000` DAI from AAVE Portal and send the `1,000,000` DAI to Alice's wallet on Polygon domain

When slow nomad message arrives, `BridgeFacet._reconcile` function is triggered to attempt to repay back the loan to AAVE portal. This function will in turn trigger the `BridgeFacet._reconcileProcessPortal` function where the portal repayment logic resides.

Within the `BridgeFacet._reconcileProcessPortal`, notice that if the `AssetLogic.swapFromLocalAssetIfNeededForExactOut` swap fails, it will return `_amount`.

Within the `BridgeFacet._reconcileProcessPortal`, notice that if the ``AavaPool.backUnbacked `external repayment call fails, it will set` amountIn = 0 `, and then return [` return (_amount - amountIn)

```
](https://github.com/code-423n4/2022-06-
connext/blob/b4532655071566b33c41eac46e75be29b4a381ed/contracts/contracts/core/connext/facets/BridgeFacet.sol
, which is basically the same as
```

_amount`.

When the `_reconcileProcessPortal` function call returned at Line 603, it will set the `toDistribute` to the `amount`. `amount` in this example is `1,000,000` DAI.

> Next, at Line 611, the contract will increase Bob's router balance by `1,000,000` DAI
>
> Bob notices that his router balance has increased by `1,000,000` DAI, and he could not resist the tempation of `1,000,000` DAI. Therefore, instead of helping Connext to repay the loan via `repayAavePortal` function out-of-band, he decided to quickly withdraws all his liquidty from his router.
>
> Bob gained `1,000,000` DAI, while Connext still owns AAVE portal `1,000,000` DAI

Impact

If routers decided not to repay the loan, Connext will incur large amount of debt from AAVE portal.

Recommended Mitigation Steps

Understood that there is a whitelist for routers that can use portals to ensure that only trusted routers could use this feature. In this case, the trust entirely depends on the integrity of the router owner and the assumption that the owner will not act against Connext and its community. However, as seen in many of the past security incidents, trusted actor or even own protocol team member might turn rogue when dealing with significant gain. In the above example, `1,000,000` DAI. It is common to see even larger amount of funds transferring across the bridge.

Therefore, to overcome the above-mentioned risk, some protocol would implement `m-of-n` multisig or validation, which help to mitigate the risk of a single trusted actor from turning rogue and perform malicious action.

Therefore, it is recommended to reconsider such design and explore other alternatives. One such alternative would be as follows:

> Assuming that the AAVE portal interest rate is fixed, therefore, the amount of repayment is deterministic, and Connext can compute the amount of repayment that needs to be repaid at any point of time.
>
> When the `BridgeFacet._reconcileProcessPortal` swap or `AavaPool.backUnbacked` fails, do not immediately credit the Bob's router balance. Instead, escrow the amount (`1,000,000` DAI) received from nomad in an `Escrow` contract. Implement a function called `settleAAVEPortalLoan` within the `Escrow` contract, which contains the logic to perform the necessary actions to repay AAVE portal loan. In this case, Bob is responsible for triggering the `Escrow.settleAAVEPortalLoan` to kick start the out-of-band repayment process. If the repayment is sucessful, Bob's router will be credited with the earning for taking on credit risk.
>
> One positive side effect of this approach is that Bob will be incentivize to make the repayment as fast as possible because the longer he delays, the higher the interest rate, and thus less earning for him.
>
> This approach is quite similar to the withdrawal pattern.

LayneHaber (Connext) acknowledged and commented:

> This is correct, and using an escrow contract would be helpful, but in general the router has no incentive ever to repay aave loans (even with this fix). This eliminates the

> possibility of a router profiting from the mishandling of `reconcile`, but doesn't address the root of the trustedness, which is embedded at the aave layer (by being able to take out unbacked loans).

[H-06] MALICIOUS RELAYER CAN REPLAY EXECUTE CALLDATA ON DIFFERENT CHAINS CAUSING DOUBLE-SPEND ISSUE

Submitted by xiaoming90

BridgeFacet.sol#L411

> This issue is only applicable for fast-transfer. Slow transfer would not have this issue because of the built-in fraud-proof mechanism in Nomad.

First, the attacker will attempt to use Connext to send `1000 USDC` from Ethereum domain to Optimism domain.

Assume that the attacker happens to be a relayer on the relayer network utilised by Connext, and the attacker's relayer happens to be tasked to relay the above execute calldata to the Optimism's Connext `BridgeFacet.execute` function.

Optimism's Connext `BridgeFacet.execute` received the execute calldata and observed within the calldata that it is a fast-transfer and Router A is responsible for providing the liquidity. It will then check that the router signature is valid, and proceed to transfer `1000 oUSDC` to attacker wallet (0x123456) in Optimism.

Next, attacker will update the `ExecuteArgs.local` within the execute calldata to a valid local representation of canonical token (USDC) used within Polygon. Attacker will then send the modified execute calldata to Polygon's Connext `BridgeFacet.execute` function. Assume that the same Router A is also providing liquidity in Polygon. The `BridgeFacet.execute` function checks that the router signature is valid, and proceed to transfer `1000 POS-USDC` to atttack wallet (0x123456) in Polygon.

At this point, the attacker has `1000 oUSDC` and `1000 POS-USDC` in his wallets. When the nomad message arrives at Optimism, Router A can claim the `1000 oUSDC` back from Connext. However, Router A is not able to claim back any fund in Polygon.

Note that same wallet address exists on different chains. For instance, the wallet address on Etherum and Polygon is the same.

Why changing the `ExecuteArgs.local` does not affect the router signature verification?

This is because the router signature is generated from the `transferId` + `pathLength` only, and these data are stored within the `CallParams params` within the `ExecuteArgs` struct.

LibConnextStorage.sol#L77

```
struct ExecuteArgs {
  CallParams params;
  address local; // local representation of canonical token
  address[] routers;
  bytes[] routerSignatures;
  uint256 amount;
  uint256 nonce;
  address originSender;
}
```

Within the `BridgeFacet._executeSanityChecks` function, it will attempt to rebuild to `transferId` by calling the following code:

```
// Derive transfer ID based on given arguments.
bytes32 transferId = _getTransferId(_args);
```

Within the `BridgeFacet._getTransferId` function, we can see that the `s.tokenRegistry.getTokenId(_args.local)` will always return the canonical `tokenDomain` and `tokenId`. In our example, it will be `Ethereum` and `USDC`. Therefore, as long as the attacker specify a valid local representation of canonical token on a chain, the `transferId` returned by `s.tokenRegistry.getTokenId(_args.local)` will always be the same across all domains. Thus, this allows the attacker to modify the `ExecuteArgs.local` and yet he could pass the router signature check.

BridgeFacet.sol#L719

```
  function _getTransferId(ExecuteArgs calldata _args) private view returns (bytes32) {
    (uint32 tokenDomain, bytes32 tokenId) = s.tokenRegistry.getTokenId(_args.local);
    return _calculateTransferId(_args.params, _args.amount, _args.nonce, tokenId, tokenDom
  }
```

Impact

Router liquidity would be drained by attacker, and affected router owner could not claim back their liquidity.

Recommended Mitigation Steps

The security of the current Connext design depends on how secure or reliable the relayer is. If the relayer turns rouge or acts against Connext, many serious consequences can happen.

The root cause is that the current design places enormous trust on the relayers to accurately and reliably to deliver calldata to the bridge in various domains. For instance, delivering of execute call data to `execute` function. There is an attempt to prevent message replay on a single domain, however, it does not prevent message replay across multiple domains. Most importantly, the Connext's bridge appears to have full trust on the calldata delivered by the relayer. However, the fact is that the calldata can always be altered by the relayer.

Consider a classic 0x off-chain ordering book protocol. A user will sign his order with his private key, and attach the signature to the order, and send the order (with signature) to the relayer network. If the relayer attempts to tamper the order message or signature, the decoded address will be different from the signer's address and this will be detected by 0x's Smart contract on-chain when processing the order. This ensures that the integrity of the message and signer can be enforced.

Per good security practice, relayer network should always be considered as a hostile environment/network. Therefore, it is recommended that similar approach could be taken with regards to passing execute calldata across domains/chains.

For instance, at a high level, the sequencer should sign the execute calldata with its private key, and attach the signature to the execute calldata. Then, submit the execute calldata (with signature) to the relayer network. When the bridge receives the execute calldata (with signature), it can verify if the decoded address matches the sequencer address to ensure that the calldata has not been altered. This will ensure the intergrity of the execute calldata and prevent any issue that arise due to unauthorised modification of calldata.

Additionally, the execute calldata should also have a field that correspond to the destination domain. The bridge that receives the execute calldata must verify that the execute calldata is intended for its domain, otherwise reject the calldata if it belongs to other domains. This also helps to prevent the attack mentioned earlier where same execute calldata can be accepted in different domains.

LayneHaber (Connext) confirmed and commented:

> Agree that this is an issue, but disagree with the framing and mitigation.
>
> The `calldata` is included in the generation of the `transferId` via the `CallParams`, so it cannot be easily manipulated by the relayer network once signed by routers. However, because you are not validating the `s.domain` against the `CallParams.destinationDomain` you can use the same transfer data across multiple chains, which is a big problem.

LayneHaber (Connext) resolved:

> connext/nxtp@bc241f8

0xleastwood (judge) commented:

> This seems like the most severe finding of the entire contest. Kudos to the warden on a great find!
>
> Because transfer data is replicated across multiple chains, relayers are also able to execute data on each chain. If `_executeSanityChecks` does not check that the message's destination chain matches `s.domain`, then transfers could be spent on all available chains.
>
> Interestingly, because the remote router is included in the message, only the correct destination chain will be able to reconcile the transfer and reimburse routers for providing liquidity. Hence, the issue is only prevalent on other chains if routers readily bid on incoming transfers, which seems possible because signatures can be replayed on other chains. So if the same set of routers have sufficient liquidity on another chain, the relayer can execute this message again to create a double spend issue.

> Another point to add, this issue would only be prevalent on chains which have its local asset pointing to the same address as this is what the bridge will attempt to transfer to the recipient. Additionally, in order for the relayer to replay a router's signature, the `transferId` must exactly match the `transferId` on the intended destination chain. This is only possible if `TokenRegistry.getTokenId` returns the same canonical domain and ID values.
>
> It would be good to confirm this. Is it possible for a local asset to be registered to the same canonical domain and ID on multiple chains?

LayneHaber (Connext) commented:

> > Is it possible for a local asset to be registered to the same canonical domain and ID on multiple chains?
>
> Yes, that is actually the purpose of the `canonicalId` and `canonicalDomain` — there should only be one canonical (locked) token that maps to any number of local (minted) instances.
>
> This issue is valid, and enforcing in `_executeSanityChecks` it is only executed on the destination domain should prevent this attack, correct?

0xleastwood (judge) commented:

> Okay great! Because local assets map to the same canonical domain and ID on each chain, I think this issue is most definitely valid. `_args.local` is not used to calculate `transferId`, hence the representation for each asset may differ on each chain but the `TokenRegistry.getTokenId` should return the correct information.
>
> I can confirm that the enforcing check in `_executeSanityChecks` should ensure that transfer data is only executed on the intended destination domain.

Medium Risk Findings (20)

[M-01] RELAYER WILL NOT RECEIVE ANY FEE IF `EXECUTE` REVERTS

Submitted by xiaoming90

Connext relies on the relayer to trigger the `BridgeFacet.execute` function on the destination domain to initiate the token transfer and calldata execution processes. Relayers pay for the gas cost to trigger the `execute` function, and in return for their effort, they are reimbused with the relayer fee.

However, it is possible that the `BridgeFacet.execute` function will revert under certain circumstances when triggered by the relayers. For instance, when the `BridgeFacet.execute` function is triggered, it will call the `BridgeFacet._handleExecuteLiquidity` function. Within the `BridgeFacet._handleExecuteLiquidity` function, it will attempt to perform token swap using a

StablePool. If the slippage during the swap exceeded the user-defined value, the swap will revert and subseqently the `execute` will revert too.

When the `BridgeFacet.execute` reverts, the relayers will not receive any relayer fee.

The following code shows that the relayer who can claim the relayer fee is set within the `BridgeFacet.execute` function at Line 415. Therefore, if this function reverts, relayer will not be able to claim the fee.

BridgeFacet.sol#L415

```
function execute(ExecuteArgs calldata _args) external whenNotPaused nonReentrant returns
  (bytes32 transferId, bool reconciled) = _executeSanityChecks(_args);

  // Set the relayer for this transaction to allow for future claim
  s.transferRelayer[transferId] = msg.sender;

  // execute router liquidity when this is a fast transfer
  // asset will be adopted unless specified to be local in params
  (uint256 amount, address asset) = _handleExecuteLiquidity(transferId, !reconciled, _ar

  // execute the transaction
  uint256 amountWithSponsors = _handleExecuteTransaction(_args, amount, asset, transferI

  // emit event
  emit Executed(transferId, _args.params.to, _args, asset, amountWithSponsors, msg.sende

  return transferId;
}
```

Impact

Loss of fund for the relayers as they pay for the gas cost to trigger the functions, but did not receive any relayer fee in return.

Recommended Mitigation Steps

Update the implementation of the `BridgeFacet.execute` so that it will fail gracefully and not revert when the swap fails or other functions fails. Relayers should be entitled to relayer fee regardless of the outcome of the `BridgeFacet.execute` call for their effort.

jakekidd (Connext) acknowledged and commented:

> It's quite possible that the slippage changes while the relayer's tx is in the mempool - which I think is the valid concern here. A relayer can't know for a fact that another existing tx won't change the current slippage (assuming there are multiple approved relayers).
>
> > Relayers should be entitled to relayer fee regardless of the outcome of the BridgeFacet.execute call for their effort.
>
> Worth noting that the mitigation step here ^ is tricky — it can incentivize relayers to submit transactions that fail quickly to maximize profit. For example, if we pay relayers even in the event of failure when slippage is too high, they are incentivized to submit txs *when* the

> slippage is too high (because they will fail more cheaply, as opposed to fully executing). Relayers could potentially profit by pushing the slippage over a large user transfer's limit via the stableswap themselves.
>
> I'm uncertain of whether I should acknowledge or dispute this issue because, while it is valid that relayers will lose funds in the case that the slippage changes while their tx is in the mempool, this may just be considered a core design property and a risk that relayers must factor into their executions.
>
> (Leaving as acknowledged for now.)

LayneHaber (Connext) commented:

> While not paying out for every relayed transaction (i.e. forcing relayers to incur/budget for some loss if the transaction fails) is a valid concern, and makes the system less appealing to relay for, I think it is the nature of relay networks to deal with these kinds of problems.
>
> For example, even without the slippage, imagine there are two networks competing to relay for the same transaction. In this case, execution would fail for the second relayer, and there would be no fees remaining to pay them for their efforts. If you want to continue adding fees for the same transaction, you would be passing this failure cost onto the user (with a more stringent liveness condition).
>
> This is a valid issue, but the fixes would introduce more complexity and edge cases than make sense to handle at this level.

0xleastwood (judge) commented:

> I'd say this is part of the risk of being a relayer but definitely worth noting so keeping it as is.

[M-02] DIAMOND UPGRADE PROPOSITION CAN BE FALSIFIED

Submitted by Czar102, also found by shenwilly

DiamondCutFacet.sol#L16-L29
LibDiamond.sol#L94-L118
LibDiamond.sol#L222-L240

Diamond is to be upgraded after a certain delay to give time to the community to verify changes made by the developers. If the proposition can be falsified, the contract admins can exploit the contract in any way of their choice.

Proof of Concept

To determine the id of the proposal, only its facet changes are hashed, skipping two critical pieces of data - the `_init` and `_calldata`. During a diamond upgrade, devs can choose what code will be executed by the contract using a delegatecall. Thus, they can make the contract perform any actions of their choice.

Recommended Mitigation Steps

Add `_init` and `_calldata` to the proposition hash.

jakekidd (Connext) confirmed and resolved:

> Resolved by connext/nxtp@63badc8

0xleastwood (judge) decreased severity to Medium and commented:

> I consider this to be severe, however, it does require a malicious or compromised governance. Because of this, I would prefer to have this downgraded to `medium` severity.

[M-03] MALICIOUS RELAYER COULD EXPLOIT SPONSOR VAULTS

Submitted by 0x52, also found by csanuragjain

SponsorVault.sol#L234-L263

Sponsor vaults drained.

Proof of Concept

`reimburseRelayerFees` uses SponsorVault funds to repay users the fees they pay to relayers. A malicious relayer could create a large number of transactions with the max reimbursed relay fee specified in SponsorVault between chains for which they relay all of them. They would receive back the relay fee from the SponsorVault and they would also get the relay fee they paid themselves.

Recommended Mitigation Steps

I don't see any way to mitigate this without substantial changes to the functionality of SponsorVault. Teams deploying SponsorVaults should be informed of potential misuse and set reimburse limits accordingly.

LayneHaber (Connext) disputed and commented:

> I'm not sure I agree with this as an issue — the relayers are whitelisted (to prevent generalized front-running, and limiting relayers to reliable relay networks).
>
> Usually any dusting systems like the `reimburseRelayerFees` do not have sybil resistance (unless you connect your socials or something similar), and this is something we will have to advertise to the people who create and fund the vault.
>
> NOTE: inspiration for this feature was taken from existing nomad dusting, which has the same potential shortcomings, see here. Generally, chains have been willing to fund these vaults even with these drawbacks. That being said, balance checks could be added to add some slight guardrails!

LayneHaber (Connext) resolved:

> connext/nxtp@be9671c

0xleastwood (judge) decreased severity to Medium and commented:

> I actually agree with the warden here. This type of attack would not be recognisable and could be disguised as general protocol activity. However, users' funds are not at direct risk, but sponsors' funds who intentionally give up these funds would be at risk of leaking value. Downgrading to `medium` risk.

[M-04] `LIBDIAMOND.DIAMONDCUT()` SHOULD CHECK `DIAMONDSTORAGE().ACCEPTANCETIMES[KECCAK256(ABI.ENCODE(_DIAMONDCUT))] != 0`

Submitted by GimelSec, also found by csanuragjain, Czar102, Lambda, minhquanym, and shenwilly

LibDiamond.sol#L100-L103
LibDiamond.sol#L71-L79
LibDiamond.sol#L83-L90

Normally, `diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))]` will be set in `LibDiamond.proposeDiamondCut()`. Then in `LibDiamond.diamondCut()`, it checks that `diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))] < block.timestamp`.

However, `LibDiamond.rescindDiamondCut()` will set `diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))]` to 0. Which can easily pass the check in `diamondCut()`. But `rescindDiamondCut` should rescind `_diamondCut`. In conclusion, using `rescindDiamondCut()` can easily bypass the delay time.

Moreover, if `proposeDiamondCut()` has never been called, the check for delay time is always passed.

Proof of Concept

`diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))]` will be set in `LibDiamond.proposeDiamondCut()`
LibDiamond.sol#L71-L79

```
function proposeDiamondCut(
  IDiamondCut.FacetCut[] memory _diamondCut,
  address _init,
  bytes memory _calldata
) internal {
  uint256 acceptance = block.timestamp + _delay;
  diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))] = acceptance;
  emit DiamondCutProposed(_diamondCut, _init, _calldata, acceptance);
}
```

Then in `LibDiamond.diamondCut()`, it checks that `diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))] < block.timestamp`
LibDiamond.sol#L100-L103

```
function diamondCut(
  IDiamondCut.FacetCut[] memory _diamondCut,
  address _init,
  bytes memory _calldata
) internal {
  require(
    diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))] < block.timesta
    "LibDiamond: delay not elapsed"
  );
  …
}
```

However, `LibDiamond.rescindDiamondCut()` will set `diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))]` to 0. Which can easily pass the check in `diamondCut()`

LibDiamond.sol#L83-L90

function rescindDiamondCut(IDiamondCut.FacetCut[] memory *diamondCut, address _init, bytes memory _calldata) internal { diamondStorage().acceptanceTimes[keccak256(abi.encode(*diamondCut))] = 0; emit DiamondCutRescinded(_diamondCut, _init, _calldata); }

```
diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))] = 0 < block.timestamp
```

Recommended Mitigation Steps

Add another check in `diamondCut`

```
function diamondCut(
  IDiamondCut.FacetCut[] memory _diamondCut,
  address _init,
  bytes memory _calldata
) internal {
  require(
    diamondStorage().acceptanceTimes[keccak256(abi.encode(_diamondCut))] < block.timesta
    "LibDiamond: delay not elapsed"
  );
  …
}
```

LayneHaber (Connext) confirmed

jakekidd (Connext) resolved:

> Resolved by connext/nxtp@cde1353

0xleastwood (judge) decreased severity to Medium and commented:

> I believe this issue to be valid but of `medium` severity as it requires a malicious or compromised governance. This issue would allow the protocol's admin to propose and execute any arbitrary data within the same transaction.

[M-05] DID NOT APPROVE TO ZERO FIRST CAUSING CERTAIN TOKEN TRANSFER TO FAIL

Submitted by xiaoming90, also found by 0xNineDec, hyh, Ruhum, and slywaters

BridgeFacet.sol#L984
AssetLogic.sol#L347

Proof-of-Concept

Some tokens (like USDT) do not work when changing the allowance from an existing non-zero allowance value. For example Tether (USDT)'s `approve()` function will revert if the current approval is not zero, to protect against front-running changes of approvals.

Instance 1 - `BridgeFacet._reconcileProcessPortal`

The following function must be approved by zero first, and then the `SafeERC20.safeIncreaseAllowance` function can be called. Otherwise, the `_reconcileProcessPortal` function will revert everytime it handles such kind of tokens. Understood from the comment that after the backUnbacked call there could be a remaining allowance.

BridgeFacet.sol#L984

```
function _reconcileProcessPortal(
  uint256 _amount,
  address _local,
  address _router,
  bytes32 _transferId
) private returns (uint256) {
      ..SNIP..
  SafeERC20.safeIncreaseAllowance(IERC20(adopted), s.aavePool, totalRepayAmount);

  (bool success, ) = s.aavePool.call(
    abi.encodeWithSelector(IAavePool.backUnbacked.selector, adopted, backUnbackedAmount,
  );
      ..SNIP..
}
```

Instance 2 - `BridgeFacet_swapAssetOut`

The following fucntion must first be approved by zero, follow by the actual allowance to be approved. Otherwise, the `_swapAssetOut` function will revert everytime it handles such kind of tokens.

AssetLogic.sol#L347

```
function _swapAssetOut(
  bytes32 _canonicalId,
  address _assetIn,
  address _assetOut,
  uint256 _amountOut,
  uint256 _maxIn
)
  internal
  returns (
    bool,
    uint256,
    address
  )
{
  AppStorage storage s = LibConnextStorage.connextStorage();

  bool success;
  uint256 amountIn;

  // Swap the asset to the proper local asset
  if (stableSwapPoolExist(_canonicalId)) {
    // get internal swap pool
    SwapUtils.Swap storage ipool = s.swapStorages[_canonicalId];
    // if internal swap pool exists
    uint8 tokenIndexIn = getTokenIndexFromStableSwapPool(_canonicalId, _assetIn);
    uint8 tokenIndexOut = getTokenIndexFromStableSwapPool(_canonicalId, _assetOut);
    // calculate slippage before performing swap
    // NOTE: this is less efficient then relying on the `swapInternalOut` revert, but ma
    // to handle slippage failures (this can be called during reconcile, so must not fai
    if (_maxIn >= ipool.calculateSwapInv(tokenIndexIn, tokenIndexOut, _amountOut)) {
      success = true;
      amountIn = ipool.swapInternalOut(tokenIndexIn, tokenIndexOut, _amountOut, _maxIn);
    }
    // slippage is too high to perform swap: success = false, amountIn = 0
  } else {
    // Otherwise, swap via stable swap pool
    IStableSwap pool = s.adoptedToLocalPools[_canonicalId];
    uint256 _amountIn = pool.calculateSwapOutFromAddress(_assetIn, _assetOut, _amountOut
    if (_amountIn <= _maxIn) {
      // set the success
      success = true;

      // perform the swap
      SafeERC20.safeApprove(IERC20(_assetIn), address(pool), _amountIn);
      amountIn = pool.swapExactOut(_amountOut, _assetIn, _assetOut, _maxIn);
    }
    // slippage is too high to perform swap: success = false, amountIn = 0
  }

  return (success, amountIn, _assetOut);
}
```

Impact

Both the `_reconcileProcessPortal` and `_swapAssetOut` functions are called during repayment to Aave Portal if the fast-transfer was executed using portal liquidity. Thus, it is core part of the token transfer process within Connext, and failure of any of these functions would disrupt the AAVE repayment process.

Since both functions affect the AAVE repayment process, I'm grouping them as one issue.

Recommended Mitigation Steps

As Connext bridges/routers deal with all sort of tokens existed in various domains/chains, the protocol should try to implement measure to ensure that it is compatible with as much tokens as possible for future growth and availability of the protocol.

Instance 1 - `BridgeFacet._reconcileProcessPortal`

It is recommended to set the allowance to zero before increasing the allowance

```
SafeERC20.safeApprove(IERC20(_assetIn), address(pool), 0);
SafeERC20.safeIncreaseAllowance(IERC20(adopted), s.aavePool, totalRepayAmount);
```

Instance 2 - `BridgeFacet_swapAssetOut`

It is recommended to set the allowance to zero before each approve call.

```
SafeERC20.safeApprove(IERC20(_assetIn), address(pool), 0);
SafeERC20.safeApprove(IERC20(_assetIn), address(pool), _amountIn);
```

ecmendenhall (Connext) commented:

> I gave this one a ❤️ for noting the special case of USDT. Since USDT's `approve` reverts if current allowance is nonzero, even calls to `safeIncreaseAllowance` must be zeroed first. Many related findings identify the same issue but recommend using `safeIncreaseAllowance` only, which would still fail in the case of USDT.

jakekidd (Connext) confirmed

LayneHaber (Connext) disagreed with severity and commented:

> This should be higher severity as the funds could get stuck.

jakekidd (Connext) resolved:

> Fixed by
> https://github.com/connext/nxtp/commit/f9819759c3915a1470a941c1b38a7d415c0dc2aa

0xleastwood (judge) commented:

> I'm actually going to disagree with the request to raise the severity of this issue because of a few main reasons:
>
> > In instance 1, the implementation for `executeBackUnbacked` transfers exactly `backUnbackedAmount + portalFee` which is equal to `totalRepayAmount`. Hence, the

> > approval is always fully utilised unless the Aave pool contract fails to execute as intended.
> >
> > In instance 2, `_swapAssetOut` explicitly approves `_amountIn` which is determined by `_calculateSwapInv`. This same function is also used to actually perform the swap, hence, this value will remain the same unless the user is able to make state changes in between these calls. It does not look like this is possible unless `safeApprove` gives the user control over contract execution.
>
> I think based on this, it is safe to say that certain assumptions about the behaviour of these contracts must be broken in order for funds to be at risk. Due to this, I think `medium` severity makes more sense.

[M-06] _HANDLEEXECUTETRANSACTION MAY NOT WORKING CORRECTLY ON FEE-ON-TRANSFER TOKENS. MOREOVER, IF IT IS FAILED, FUND MAY BE LOCKED FOREVER.

Submitted by Chom, also found by csanuragjain

BridgeFacet.sol#L856-L877
Executor.sol#L142-L144
Executor.sol#L160-L166
Executor.sol#L194-L213

`_handleExecuteTransaction` may not working correctly on fee-on-transfer tokens. As duplicated fee is applied to fee on transfer token when executing a arbitrary call message passing request. Moreover, the Executor contract increase allowance on that token for that target contract in full amount without any fee, this may open a vulnerability to steal dust fund in the contract

Moreover, failure is trying to send full amount without any fee which is not possible because fee is already applied one time for example 100 Safemoon -> 90 Safemoon but trying to transfer 100 safemoon to `_recovery address`. Obviously not possible since we only have 90 Safemoon. This will revert and always revert causing loss of fund in all case of fee-on-transfer tokens.

Proof of Concept

A message to transfer 100 Safemoon with some contract call payload has been submitted by a relayer to `_handleExecuteTransaction` function on BridgeFacet these lines hit.

```
// execute calldata w/funds
AssetLogic.transferAssetFromContract(_asset, address(s.executor), _amount);
(bool success, bytes memory returnData) = s.executor.execute(
  IExecutor.ExecutorArgs(
    _transferId,
    _amount,
    _args.params.to,
    _args.params.recovery,
    _asset,
    _reconciled
```

```
      ? LibCrossDomainProperty.formatDomainAndSenderBytes(_args.params.originDomain,
      : LibCrossDomainProperty.EMPTY_BYTES,
    _args.params.callData
  )
);
```

Noticed that 100 Safemoon is transferred to executor before contract execution. Now executor has 90 Safemoon due to 10% fee on transfer.

```
if (!isNative) {
  SafeERC20Upgradeable.safeIncreaseAllowance(IERC20Upgradeable(_args.assetId), _args.t
}
```

Next, we increase allowance of target contract to transfer 100 more Safemoon.

```
// 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
);
```

After that, it call target contract

```
// 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
);
```

Target contract tried to pull 100 Safemoon from Executor but now Executor only has 90 Safemoon, so contract call failed. Moving on to the failure handle.

```
function _handleFailure(
  bool isNative,
  bool hasIncreased,
  address _assetId,
  address payable _to,
  address payable _recovery,
  uint256 _amount
) private {
  if (!isNative) {
    // Decrease allowance
    if (hasIncreased) {
      SafeERC20Upgradeable.safeDecreaseAllowance(IERC20Upgradeable(_assetId), _to, _amou
    }
    // Transfer funds
    SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(_assetId), _recovery, _amount);
  } else {
    // Transfer funds
    AddressUpgradeable.sendValue(_recovery, _amount);
  }
}
```

isNative = false because of Safemoon

Trying to transfer 100 Safemoon to _recovery while only having 90 Safemoon in the contract. Thus failure handle is reverted.

Recommended Mitigation Steps

You should approve one step only. Avoid an extra token transfer.

LayneHaber (Connext) confirmed and commented:

> Not sure I agree with the mitigation (perhaps a better alternative would be to transfer the balance of the executor), but understand the problems!

LayneHaber (Connext) acknowledged and commented:

> Upon further reflection, choosing to remove support for fee on transfer tokens. The problems are much deeper than just issues on `execute` — the minted token will *not* have fees on transfers (uses vanilla ERC20 implementation), and i doubt the `StableSwap` implementations are taking these into account as well. Changing label to "acknowledged"!

LayneHaber (Connext) resolved:

> error on fee: connext/nxtp@14df4c6

0xleastwood (judge) decreased severity to Medium and commented:

> The destination chain will indeed be unable to handle bridge transfers involving fee-on-transfer tokens. Although, its worth adding that this is only made possible because `handleIncomingAsset` and `swapToLocalAssetIfNeeded` in `xcall` do not fail when the provided asset has some fee-on-transfer behaviour.
>
> Because `_args.amount` is not overridden with the actual `amount` transferred in from `handleIncomingAsset`, routers on the destination chain will attempt to provide liquidity for the amount transferred by the user + the fee. Furthermore, when the transfer has been fully bridged, routers who fronted the liquidity will receive less funds than expected. However, I don't actually think this issue warrants `high` severity, mainly because the bridge transfer should actually execute successfully.
>
> The only issue is that if routers front liquidity, they are exposing themselves to receiving slightly less funds due to the fee upon reconciliation. Hence, only value is being leaked

and I think `medium` severity makes more sense.

[M-07] CURRENT IMPLEMENTATION OF ARBITRARY CALL EXECUTE FAILURE HANDLER MAY BREAK SOME USE CASE FOR EXAMPLE NFT BRIDGE.

Submitted by Chom

Executor.sol#L113-L192
Executor.sol#L194-L213

Current implementation of arbitrary call execute failure handler may break some use case for example NFT Bridge.

In the case of NFT Bridge, NFT may be lost forever.

This is likely to be happened in the case of out of gas.

Proof of Concept

Relayer receive the message to unlock BAYC on ETH chain. Relayer call execute on BridgeFacet which then call execute in Executor internally. Continue to these lines:

```
// Ensure there is enough gas to handle failures
uint256 gas = gasleft() - FAILURE_GAS;

// 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
);

// Unset properties
properties = LibCrossDomainProperty.EMPTY_BYTES;

// Unset amount
amnt = 0;

// Handle failure cases
if (!success) {
  _handleFailure(isNative, true, _args.assetId, payable(_args.to), payable(_args.recov
}
```

Unfortunately, an enormous NFT project has just started minting their NFT when the relayer perform the execution. Causing gas price to increase by 20x.

As a result, gasLimit is 20x lesser than what we have calculated causing ExcessivelySafeCall.excessivelySafeCall to fail because of out of gas. We fall into _handleFailure function. Notice that NFT is not unlocked yet because target contract has failed to being executed.

```
function _handleFailure(
  bool isNative,
  bool hasIncreased,
  address _assetId,
  address payable _to,
  address payable _recovery,
  uint256 _amount
) private {
  if (!isNative) {
    // Decrease allowance
    if (hasIncreased) {
      SafeERC20Upgradeable.safeDecreaseAllowance(IERC20Upgradeable(_assetId), _to, _amou
    }
    // Transfer funds
    SafeERC20Upgradeable.safeTransfer(IERC20Upgradeable(_assetId), _recovery, _amount);
  } else {
    // Transfer funds
    AddressUpgradeable.sendValue(_recovery, _amount);
  }
}
```

`_handleFailure` just sent dummy fund in the NFT bridge process to the useless fallback address (Useless in NFT bridge case as it doesn't involve any cross chain swapping / token transferring).

Finally, transferId will be marked as used (or reconciled). This transferId cannot be used anymore.

Recall that BAYC hasn't been unlocked yet as target contract has failed to being executed.

And we cannot reuse this transferId to retry anymore in the future as it is marked as used

As a result, BAYC is locked forever as target contract call never success anymore since transferId has been marked as used

Recommended Mitigation Steps

You should mix axelar style and connext style of error handling together.

Fund shouldn't be sent to fallback address immediately. Instead, leave an option for user to choose whether they want to recall the failed transaction manually or they want to transfer the fund to the fallback address.

LayneHaber (Connext) acknowledged and commented:

> Acknowledge that this should be an issue, but think that the UX of having to submit multiple destination chain transactions is not great (and causes problems with fees / assuming the user has gas on the destination domain). I think this type of thing needs to be handled by the integrator themselves

Chom (warden) commented:

> I have reviewed this once again and found another case that current implementation of arbitrary call execute failure handler may break some use case for example NFT bridge.

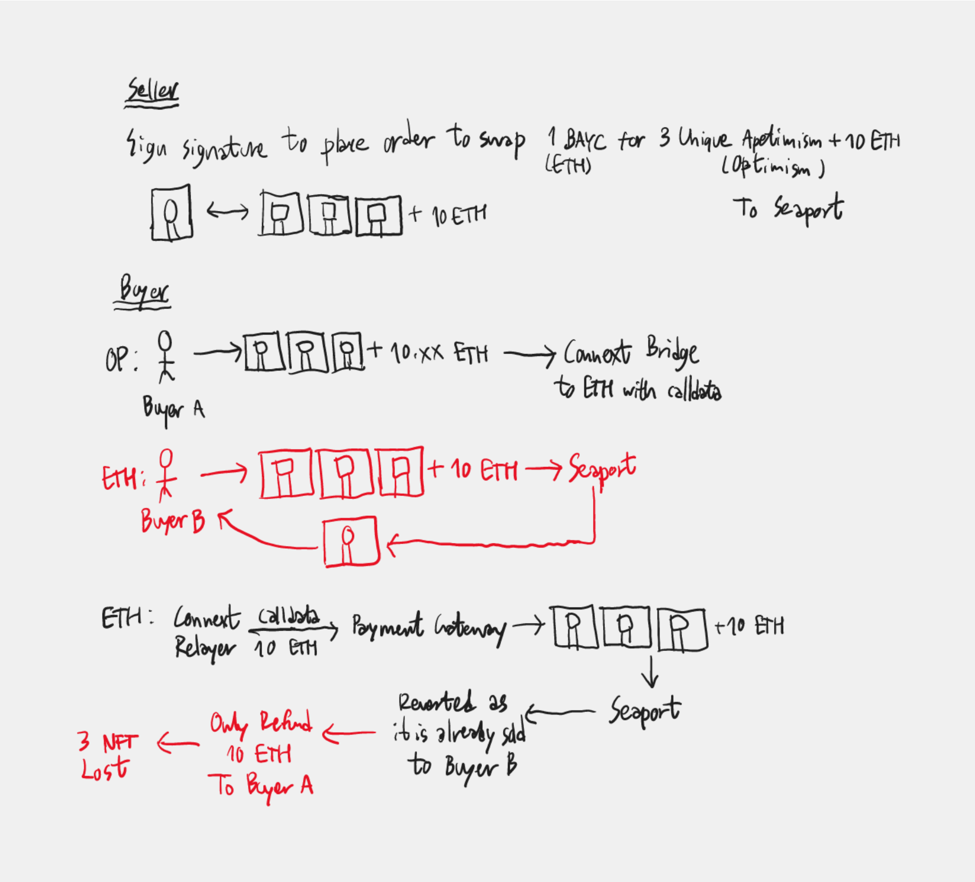
> The idea for this to happen is simply do anything to set `success` to false. If NFT is failed to mint on the destination chain and revert (Likely due to bug in the contract), it cause the same consequence.
>
> It would be better to revert and allow it to be executed again in the future by simply revert in case of arbitrary call failed. If the destination contract is upgradable, it can be fixed but if not it is out of luck. But current implementation is out of luck in all case.
>
> Anyway, the sudden gas price increase problem can't be handled accurately as its may happened very fast. (5 gwei 2 minutes ago, 50 gwei when the NFT unlock as a big NFT project is minting). No one will spare that much gas while locking NFT in the source chain 2 minutes ago.

0xleastwood (judge) decreased severity to Medium and commented:

> Gas limit does not change deterministically. If you make a call which provides X amount of gas at Y price, if gas price changes, it will not affect any pending calls. The issue only arises if the relayer intentionally provides insufficient gas at any price.
>
> The recovery address is configured when the bridge transfer is initiated, if this or the executed calldata fails to be used correctly, I think the onus is on the sender. However, there are some legitimate concerns of a bridge relayer intentionally making the callback fail. Because funds aren't at direct risk, `medium` severity would make more sense.

Chom (warden) commented:

> The recovery address only recover ERC20 tokens managed by the Connext bridge system. Other custom made bridging solution that utilize the arbitrary message passing never utilize that recovery address passed.
>
> ### Example
>
> For example, if A lock "multiple BAYC NFT" on the Ethereum chain along with 1 WETH sent into Connext Amarok. On the destination chain, if executed calldata fails or gas limit is miscalculated (both intentionally and non-intentionally in case ether.js returns too low gas limit, it has happened many times on the complex contracts in 2021 not sure it is fixed or not), only 1 WETH is sent to the recovery address and not "multiple BAYC NFT". "Multiple BAYC NFT" will be locked on the Ethereum chain indefinitely without any chance to unlock or recovery it in the source chain. So, funds which are "multiple BAYC NFT" are at direct risk. BAYC floor price is currently more than 70 ETH. If 3 BAYC lost due to a single error, total 200+ ETH will be lost.
>
> Here is an example case that might become common if NFT is not out of hyped but sadly the hype in NFT is currently down. It is "NFT buying frontrunning" problem.



Since Connext bridge requires 2-3 minutes between that times somebody may frontrun buying NFT without using bridge. Then the execution will be fail since NFT has been sold to buyer B. It only refund 10 ETH but not 3 NFT sent.

Mitigation

> Simply revert in case of arbitrary call failed.
>
> Provide a way for user to cancel the execution, get ERC20 refund into the recovery address and acknowledge contract in the source chain that the bridge is failed to claim back NFT.
>
> Force all developers that are using your product to use Upgradeable contract on both source and destination chain to handle cases manually in case something went wrong (Not possible to force everyone).

Perfect mitigation is not found yet. There is an ongoing debate about this one at code-423n4/2022-07-axelar-findings#97.

0xleastwood (judge) commented:

> Fully agree with your take on this. But I still stand by my decision on keeping this as a `medium` risk issue. Mainly because funds are not at direct risk and when they are at risk, certain assumptions must be made for this to hold true. Again, the onus seems to be on the implementer's end.
>
> The important thing is to not integrate with Connext with the guarantee that calldata will

> be successfully executed on the destination contract. This should be clearly documented and I think allowing the bridge user to re-execute calldata upon failure is a potentially useful suggestion, but this doesn't come with its pitfalls too. Added complexity could expand Connext's attack surface.
>
> In response to your proposed mitigation(s):
>
> > This would impact the liveness of bridge transfers. We do not want to allow funds to get stuck due to a reliance on external requirements.
> >
> > While this is a potential solution, I don't think Connext needs to make changes to accommodate this behaviour. The destination contract can check if the calldata was successfully executed and handle this on their end. Seaport already allows for custom behaviour.
> >
> > Again, this would not require Connext to make any changes.

[M-08] MALICIOUS RELAYER COULD CAUSE A ROUTER TO PROVIDE MORE LIQUIDITY THAN IT SHOULD

Submitted by xiaoming90

Assume this is a fast-transfer path and the sequencer has a good reason (e.g. some sophisticated liquidity load balancing algorithm) to assign 3 routers to provide liquidity for a transfer of `90 DAI`

Therefore, each of them will provide `30 DAI` equally.

> _args.routers[] array = [Router A, Router B, Router C]

However, a malicious relayer could rearrange the `_args.routers[] array` to any of the following and still pass the sanity checks within `BridgeFacet._executeSanityChecks`

> _args.routers[] array = [Router A, Router A, Router A]

> _args.routers[] array = [Router A, Router A, Router B]

> _args.routers[] array = [Router A, Router A, Router C]

> _args.routers[] array = [Router C, Router A, Router C]

The point is that as long as the attacker ensures that the `pathLength` is correct, he will be able to pass the router check within `BridgeFacet._executeSanityChecks`.

Assume that malicious relayer decided to rearrange the `_args.routers[] array` to as follows, this will cause Router A to provide more liquidity than it should be and overwrite the sequencer decision. In this case, Router A will be forced to provide `90 DAI`.

> _args.routers[] array = [Router A, Router A, Router A]

This is possible because the `routerHash` used to generate the router signature only consists of two items (`transferId` and `pathLength`)

BridgeFacet.sol#L636

```
function _executeSanityChecks(ExecuteArgs calldata _args) private view returns (bytes32,
  ..SNIP..
  // Derive transfer ID based on given arguments.
  bytes32 transferId = _getTransferId(_args);

  // Retrieve the reconciled record. If the transfer is `forceSlow` then it must be reco
  // before it's executed.
  bool reconciled = s.reconciledTransfers[transferId];
  if (_args.params.forceSlow && !reconciled) revert BridgeFacet__execute_notReconciled()

  // Hash the payload for which each router should have produced a signature.
  // Each router should have signed the `transferId` (which implicitly signs call params
  // amount, and tokenId) as well as the `pathLength`, or the number of routers with whi
  // they are splitting liquidity provision.
  bytes32 routerHash = keccak256(abi.encode(transferId, pathLength));

  // check the reconciled status is correct
  // (i.e. if there are routers provided, the transfer must *not* be reconciled)
  if (pathLength > 0) // make sure routers are all approved if needed
  {
    if (reconciled) revert BridgeFacet__execute_alreadyReconciled();

    for (uint256 i; i < pathLength; ) {
      // Make sure the router is approved, if applicable.
      // If router ownership is renounced (_RouterOwnershipRenounced() is true), then th
      // no longer applies and we can skip this approval step.
      if (!_isRouterOwnershipRenounced() && !s.routerPermissionInfo.approvedRouters[_arg
        revert BridgeFacet__execute_notSupportedRouter();
      }

      // Validate the signature. We'll recover the signer's address using the expected p
      // signature scheme recovery. The address for each signature must match the router
      if (_args.routers[i] != _recoverSignature(routerHash, _args.routerSignatures[i]))
        revert BridgeFacet__execute_invalidRouterSignature();
      }

      unchecked {
        i++;
      }
    }
  ..SNIP..
}
```

Impact

Malicious relayer could overwrite sequencer decision, or cause a certain router to drain more liquidity than it should be.

Recommended Mitigation Steps

Generate the `routerHash` with the following items should help to prevent this attack:

transferId

pathLength

_args.routers[] array

In this case, if the attacker attempts to re-arrange the `_args.routers[] array`, the `routerHash` generate on the bridge will be different. Thus, it will fail the router signature verification within `_executeSanityChecks` function and it will revert.

jakekidd (Connext) disagreed with severity

jakekidd (Connext) acknowledged and commented:

> The relayer doing this would actually not be an attack on the router providing liquidity; it would actually benefit that router, giving it the full bulk of the transfer - assuming the router can afford it - as that router will claims fees for the whole thing. However, it would grief other routers from getting access to this transfer in the process. This isn't great, but keep in mind relayers are currently a permissioned role (whitelisted) because the trust vectors there have not been abstracted entirely (among other reasons). ~~As such, I am suggesting we de-escalate this issue to QA/Low Risk~~.
>
> Additionally, the suggested mitigation step is invalid within current design - `routerHash` for the router's signature is generated prior to knowing which routers will be selected by sequencer. The only way to prevent this would be to check to make sure there are no duplicates in the routers array manually (which would incur not-so-great gas costs). Acknowledging the issue as we may want to implement that fix in the future.
>
> EDIT: Removed disagree with severity tag. Seems appropriate in hindsight as it would be subverting the functionality of the protocol.

0xleastwood (judge) commented:

> I agree with the validity of this finding. Keeping it as is.

[M-09] MALICIOUS RELAYERS COULD FAVOR THEIR ROUTERS

Submitted by xiaoming90, also found by csanuragjain

Assume that a malicious relayer operates a router in Connext providing fast-liquidity service. A malicious relayer could always swap the router(s) within the execute calldata with the router(s) owned by malicious relayer, and submit it to the chain for execution.

Proof-of-Concept

This example assumes a fast-liquidity path. When the relayer posts a execute calldata to destination domain's `BridgeFacet.execute` function, this function will trigger the `BridgeFacet._executeSanityChecks` function to perform a sanity check.

Assume that the execute calldata only have 1 router selected. A malicious relayer could perform the following actions:

> Attacker removes original router from `_args.routers[]` array, and remove original router's signature from `_args.routerSignatures[]` array from the execute calldata.
>
> Attacker re-calculates the routerHash (`routerHash = keccak256(abi.encode(transferId, pathLength))`). pathLength = 1 in this

example

Attacker signs the routerHash with his own router's private key to obtain the router signature

Attacker inserts his router to the `_args.routers[]` array, and insert the router signature obtained fromthe previous step to `_args.routerSignatures[]` array

Submit the modified execute calldata to destination domain's `BridgeFacet.execute` function, and it will pass the `BridgeFacet._executeSanityChecks` function since router signature is valid.

The `BridgeFacet._executeSanityChecks` function is not aware of the fact that the router within the execute calldata has been changed because it will only check if the router specified within the `_args.routers[]` array matches with the router signature provided. Once the sanity check passes, it will store the attacker's router within `s.routedTransfers[_transferId]` and proceed with providing fast-liquidity service for the users.

The existing mechanism is useful in preventing malicious relayer from specifying routers belonging to someone else because the malicious relayer would not be capable of generating a valid router signature on behalf of other routers because he does not own their private key. However, this mechanism does not guard against a malicious relayer from specifying their own router because in this case they would be able to generate a valid router signature as they own the private key.

BridgeFacet.sol#L636

```
/**
  * @notice Performs some sanity checks for `execute`
  * @dev Need this to prevent stack too deep
  */
 function _executeSanityChecks(ExecuteArgs calldata _args) private view returns (bytes32,
   // If the sender is not approved relayer, revert
   if (!s.approvedRelayers[msg.sender] && msg.sender != _args.params.agent) {
     revert BridgeFacet__execute_unapprovedSender();
   }

   // Path length refers to the number of facilitating routers. A transfer is considered
   // if multiple routers provide liquidity (in even 'shares') for it.
   uint256 pathLength = _args.routers.length;

   // Make sure number of routers is below the configured maximum.
   if (pathLength > s.maxRoutersPerTransfer) revert BridgeFacet__execute_maxRoutersExceed

   // Derive transfer ID based on given arguments.
   bytes32 transferId = _getTransferId(_args);

   // Retrieve the reconciled record. If the transfer is `forceSlow` then it must be recc
   // before it's executed.
   bool reconciled = s.reconciledTransfers[transferId];
   if (_args.params.forceSlow && !reconciled) revert BridgeFacet__execute_notReconciled()

   // Hash the payload for which each router should have produced a signature.
   // Each router should have signed the `transferId` (which implicitly signs call params
   // amount, and tokenId) as well as the `pathLength`, or the number of routers with whi
   // they are splitting liquidity provision.
   bytes32 routerHash = keccak256(abi.encode(transferId, pathLength));

   // check the reconciled status is correct
   // (i.e. if there are routers provided, the transfer must *not* be reconciled)
   if (pathLength > 0) // make sure routers are all approved if needed
   {
     if (reconciled) revert BridgeFacet__execute_alreadyReconciled();

     for (uint256 i; i < pathLength; ) {
       // Make sure the router is approved, if applicable.
       // If router ownership is renounced (_RouterOwnershipRenounced() is true), then th
       // no longer applies and we can skip this approval step.
       if (!_isRouterOwnershipRenounced() && !s.routerPermissionInfo.approvedRouters[_arg
         revert BridgeFacet__execute_notSupportedRouter();
       }

       // Validate the signature. We'll recover the signer's address using the expected p
       // signature scheme recovery. The address for each signature must match the router
       if (_args.routers[i] != _recoverSignature(routerHash, _args.routerSignatures[i]))
         revert BridgeFacet__execute_invalidRouterSignature();
       }

       unchecked {
         i++;
       }
     }
   ..SNIP..
 }
```

When the nomad message eventually reaches the destination domain and triggered to the `BridgeFacet._reconcile`, the attacker's router will be able to claim back the asset provided in the execution step as per normal.

Impact

Malicious relayer could force Connext to use those routers owned by them to earn the liquidity fee, and at the same time causes the original router chosen by the sequencer to lost the opportunity to earn the liquidity fee. This disrupts the balance and fairness of the protocol causing normal routers to lost the opportunity to earn liquidity fee.

In bridge or cross-chain communication design, it is a good security practice to minimize the trust that Connext places on other external protocol (e.g. relayer network) wherever possible so that if the external protocol is compromised or acting against maliciously against Connext, the impact or damage would be reduced.

Recommended Mitigation Steps

It is recommended to devise a way for the Connext's destination bridge to verify that the execute calldata received from the relayer is valid and has not been altered. Ideally, the hash of the original execute calldata sent by seqencer should be compared with the hash of the execute calldata received from relayer so that a mismatch would indicate that the calldata has been modified along the way, and some action should be taken.

For instance, consider a classic 0x off-chain ordering book protocol. A user will sign his order with his private key, and attach the signature to the order, and send the order (with signature) to the relayer network. If the relayer attempts to tamper the order message or signature, the decoded address will be different from the signer's address and this will be detected by 0x's Smart contract on-chain when processing the order. This ensures that the integrity of the message and signer can be enforced.

Per good security practice, relayer network should always be considered as a hostile environment/network. Therefore, it is recommended that similar approach could be taken with regards to passing execute calldata across domains/chains.

For instance, at a high level, the sequencer should sign the execute calldata with its private key, and attach the signature to the execute calldata. Then, submit the execute calldata (with signature) to the relayer network. When the bridge receives the execute calldata (with signature), it can verify if the decoded address matches the sequencer address to ensure that the calldata has not been altered. This will ensure the intergrity of the execute calldata and prevent any issue that arise due to unauthorised modification of calldata.

Alternative Solution

Alternatively, following method could also be adopted to prevent this issue:

> Assume that it is possible to embed the selected router(s) within the slow nomad message. Append the selected router(s) within the slow nomad message
>
> Within the `BridgeFacet.execute` function, instead of using only the transfer ID as the array index (`s.routedTransfers[_transferId] = _args.routers;`), use both transfer ID + selected router as the array index (`s.routedTransfers[hash(_transferId+routers)] = _args.routers;`)
>
> When the slow nomad message arrives and triggers to the `BridgeFacet._reconcile`, this function will find the routers that provide the fast-liquidity based on the information within the nomad message only. It will attempt to call `s.routedTransfers[hash(_transferId+routers)]` and it should return nothing as there is a mismatch between attacker's router and router within the nomad message.
>
> In this case, the attacker will not be able to claim back any of the funds he provided earlier. This will deter anyone from attempting to swap the router within the execute calldata sent to destination domain's `BridgeFacet.execute` function because they will not be able to claim back the funds

jakekidd (Connext) commented:

> Duplicate of M-08 (Issue #149)

0xleastwood (judge) commented:

> I believe this finding is distinct from M-08 because this outlines how relayers could collude with routers to earn most of the fees for providing liquidity to bridge users. M-08 describes how a relayer may force any arbitrary router to supply more liquidity than they originally intended. The later is bad UX for routers providing liquidity even if they do earn more in fees.

[M-10] ROUTER OWNER COULD BE RUGGED BY ADMIN

Submitted by xiaoming90, also found by unforgiven

Assume that Alice's router has large amount of liquidity inside.

Assume that the Connext Admin decided to remove a router owned by Alice. The Connext Admin will call the `RoutersFacet.removeRouter` function, and all information related to Alice's router will be erased (set to 0x0) from the `s.routerPermissionInfo`.

RoutersFacet.sol#L293

```
function removeRouter(address router) external onlyOwner {
  // Sanity check: not empty
  if (router == address(0)) revert RoutersFacet__removeRouter_routerEmpty();

  // Sanity check: needs removal
  if (!s.routerPermissionInfo.approvedRouters[router]) revert RoutersFacet__removeRouter

  // Update mapping
  s.routerPermissionInfo.approvedRouters[router] = false;

  // Emit event
  emit RouterRemoved(router, msg.sender);

  // Remove router owner
  address _owner = s.routerPermissionInfo.routerOwners[router];
  if (_owner != address(0)) {
    emit RouterOwnerAccepted(router, _owner, address(0));
    // delete routerOwners[router];
    s.routerPermissionInfo.routerOwners[router] = address(0);
  }

  // Remove router recipient
  address _recipient = s.routerPermissionInfo.routerRecipients[router];
  if (_recipient != address(0)) {
    emit RouterRecipientSet(router, _recipient, address(0));
    // delete routerRecipients[router];
    s.routerPermissionInfo.routerRecipients[router] = address(0);
  }

  // Clear any proposed ownership changes
  s.routerPermissionInfo.proposedRouterOwners[router] = address(0);
  s.routerPermissionInfo.proposedRouterTimestamp[router] = 0;
}
```

Alice is aware that her router has been removed by Connext Admin, so she decided to withdraw the liquidity from her previous router by calling `RoutersFacet.removeRouterLiquidityFor`.

However, when Alice called the `RoutersFacet.removeRouterLiquidityFor` function, it will revert every single time. This is because the condition `msg.sender != getRouterOwner(_router)` will always fail.

RoutersFacet.sol#L490

```
/**
 * @notice This is used by any router owner to decrease their available liquidity for a
 * @param _amount - The amount of liquidity to remove for the router
 * @param _local - The address of the asset you're removing liquidity from. If removing
 * native asset, routers may use `address(0)` or the wrapped asset
 * @param _to The address that will receive the liquidity being removed
 * @param _router The address of the router
 */
function removeRouterLiquidityFor(
  uint256 _amount,
  address _local,
  address payable _to,
  address _router
) external nonReentrant whenNotPaused {
  // Caller must be the router owner
  if (msg.sender != getRouterOwner(_router)) revert RoutersFacet__removeRouterLiquidityF
  // Remove liquidity
  _removeLiquidityForRouter(_amount, _local, _to, _router);
}
```

Since the `RoutersFacet.removeRouter` function has earlier erased all information related to Alice's router within `s.routerPermissionInfo`, the `getRouterOwner` function will always return the router address.

In this case, the router address will not match against `msg.sender` address/Alice address, thus Alice attempts to call `removeRouterLiquidityFor` will always revert.

RoutersFacet.sol#L212

```
function getRouterOwner(address _router) public view returns (address) {
  address _owner = s.routerPermissionInfo.routerOwners[_router];
  return _owner == address(0) ? _router : _owner;
}
```

Impact

Router owner who provides liquidity could be rugged by Connext admin. When this happen, the router owner funds will be struck within the `RoutersFacet` contract, and there is no way for the router owner to retrieve their liquidity.

In the worst case scenario, a compromised Connext admin could remove all routers, and cause all liquidity to be struck within `RoutersFacet` and no router owner could withdraw their liquidity from the contract. Next, the `RouterFacet` contract could be upgraded to include additional function to withdraw all liquidity from the contract to an arbitrary wallet address.

Recommended Mitigation Steps

The router owner is still entitled to their own liquidity even though their router has been removed by Connext Admin. Thus, they should be given the right to take back their liquidity when such an event happens. The contract should update its implementation to support this. This will give more assurance to the router owner.

jakekidd (Connext) acknowledged and commented:

> The language here is slightly exaggerated - router funds could not be stolen by the Owner; the Owner can only prevent the router from accessing them by revoking their approval.
>
> However, it's still a valid concern, and one that will ultimately be addressed once router permissioning/whitelisting is removed permanently *or* the Owner role will be delegated to governance.
>
> The problem with the mitigation step proposed here is that the owner might not be set in some cases, so it's not a complete solution. So even if router approval is revoked, but we

> leave the assignment in the ownership mapping, in cases where the router did not assign an owner they won't be able to withdraw.
>
> There might be a better solution here by leaving the recipient in the corresponding mapping instead. In the `removeLiquidityFor` function, we can handle the case where the router's approval/ownership has been revoked by sending the funds to the recipient regardless of the caller. Unlike the owner, the recipient is always set on router registration.

jakekidd (Connext) confirmed and commented:

> Changing this to be confirmed - would like to resolve this issue by carrying out the solution described in above comment ^

0xleastwood (judge) commented:

> I think this is only a valid `medium` because the admin is an EOA and not delegated to a governance contract. Keeping it as is.

[M-11] TOKENS WITH DECIMALS LARGER THAN 18 ARE NOT SUPPORTED

Submitted by WatchPug, also found by 0x1f8b

For tokens with decimals larger than 18, many functions across the codebase will revert due to underflow.

ConnextPriceOracle.sol#L99-L115

```
function getPriceFromDex(address _tokenAddress) public view returns (uint256) {
    PriceInfo storage priceInfo = priceRecords[_tokenAddress];
    if (priceInfo.active) {
      uint256 rawTokenAmount = IERC20Extended(priceInfo.token).balanceOf(priceInfo.lpToken
      uint256 tokenDecimalDelta = 18 - uint256(IERC20Extended(priceInfo.token).decimals())
      uint256 tokenAmount = rawTokenAmount.mul(10**tokenDecimalDelta);
      uint256 rawBaseTokenAmount = IERC20Extended(priceInfo.baseToken).balanceOf(priceInfo
      uint256 baseTokenDecimalDelta = 18 - uint256(IERC20Extended(priceInfo.baseToken).dec
      uint256 baseTokenAmount = rawBaseTokenAmount.mul(10**baseTokenDecimalDelta);
      uint256 baseTokenPrice = getTokenPrice(priceInfo.baseToken);
      uint256 tokenPrice = baseTokenPrice.mul(baseTokenAmount).div(tokenAmount);

      return tokenPrice;
    } else {
      return 0;
    }
  }
```

StableSwapFacet.sol#L426

```
precisionMultipliers[i] = 10**uint256(SwapUtils.POOL_PRECISION_DECIMALS - decimals[i]);
```

Chainlink feeds' with decimals > 18 are not supported neither:

ConnextPriceOracle.sol#L122-L140

```
  function getPriceFromChainlink(address _tokenAddress) public view returns (uint256) {
    AggregatorV3Interface aggregator = aggregators[_tokenAddress];
    if (address(aggregator) != address(0)) {
      (, int256 answer, , , ) = aggregator.latestRoundData();

      // It's fine for price to be 0. We have two price feeds.
      if (answer == 0) {
        return 0;
      }

      // Extend the decimals to 1e18.
      uint256 retVal = uint256(answer);
      uint256 price = retVal.mul(10**(18 - uint256(aggregator.decimals())));

      return price;
    }

    return 0;
  }
```

Recommended Mitigation Steps

Consider checking if decimals > 18 and normalize the value by div the decimals difference.

ecmendenhall (Connext) commented:

> I gave this a ❤️ along with issue #61 because these findings both identified an additional location in the `StableSwap` contract where the 18 decimal assumption is hardcoded.

jakekidd (Connext) confirmed and commented:

> > I gave this a ❤️ along with #61 because these findings both identified an additional location in the `StableSwap` contract where the 18 decimal assumption is hardcoded.
>
> Marking as confirmed (and leaving issue open) for this reason. Would be great to merge both findings into 1 issue in the finalized audit.

jakekidd (Connext) commented:

> Assortment of findings across these three issues:
> https://github.com/code-423n4/2022-06-connext-findings/issues/39
> https://github.com/code-423n4/2022-06-connext-findings/issues/61
> https://github.com/code-423n4/2022-06-connext-findings/issues/204

jakekidd (Connext) resolved:

> Fixed by connext/nxtp@f2e5b66

0xleastwood (judge) commented:

> Marking this as the primary issue because it highlights an active part of the codebase while other issues do not. `initializeSwap` will not be compatible with any token with `decimals` greater than `18`.

[M-12] INCORRECT ADOPTED MAPPING ON UPDATING WRAPPER TOKEN

Submitted by codexploder

AssetFacet.sol#L100

> Admin can call setWrapper function to setup a new wrapper Y instead of old wrapper X
>
> This becomes a problem for any old asset which was setup during setupAsset call where `s.canonicalToAdopted[_canonical.id]` will still point to old wrapper X instead of Y

Recommended Mitigation Steps

If wrapper is changed then all variables storing this wrapper should also update.

jakekidd (Connext) confirmed and commented:

> Good spot. This line in `setupAsset` suggests that we should be setting the wrapper for `canonicalToAdopted`, so this is correct:
>
> AssetFacet.sol#L143
>
> Fixing this won't be super straightforward - we'll have to add another property to tell us the canonical ID(s?) that are currently pointing to the wrapper contract as the adopted asset.

0xleastwood (judge) commented:

> Agree that this would cause compatibility issues. The admin will be unable to call `setupAsset` and update the adopted mapping correctly.

[M-13] MISSING `WHENNOTPAUSED` MODIFIER

Submitted by SmartSek, also found by csanuragjain and hansfriese

StableSwapFacet.sol#L279-L286

In `StableSwapFacet.sol`, two swapping functions contain the `whenNotPaused` modifier while `swapExactOut()` and `addSwapLiquidity()` do not. All functions to swap and add liquidity should contain the same modifiers to stop transactions while paused.

Proof of Concept

Example with modifier

```
function swapExact(
  bytes32 canonicalId,
  uint256 amountIn,
  address assetIn,
  address assetOut,
  uint256 minAmountOut,
  uint256 deadline
) external payable nonReentrant deadlineCheck(deadline) whenNotPaused returns (uint256)
```

Examples without modifier

```
function swapExactOut(
  bytes32 canonicalId,
  uint256 amountOut,
  address assetIn,
  address assetOut,
  uint256 maxAmountIn,
  uint256 deadline
) external payable nonReentrant deadlineCheck(deadline) returns (uint256) {
```

and

```
function addSwapLiquidity(
  bytes32 canonicalId,
  uint256[] calldata amounts,
  uint256 minToMint,
  uint256 deadline
) external nonReentrant deadlineCheck(deadline) returns (uint256) {
  return s.swapStorages[canonicalId].addLiquidity(amounts, minToMint);
}
```

Recommended Mitigation Steps

Add the `whenNotPaused` modifier to all functions that perform swaps or liquidity additions.

jakekidd (Connext) confirmed and resolved:

> Resolved by: connext/nxtp@1dd5559

0xleastwood (judge) commented:

> I think this makes sense to add!

[M-14] SINGLE ERROR WITHIN SPONSORVAULT CONTRACT COULD CAUSE ENTIRE CROSS-CHAIN COMMUNICATION TO BREAK DOWN

Submitted by xiaoming90, also found by shenwilly

A third party sponsor would need to implement a `SponsorVault` contract that is aligned with the `ISponsorVault` interface.

Assume that a `SponsorVault` contract has been defined on Optimism chain. All cross-chain communications are required to call the `BridgeFacet.execute`, which in turn will trigger the `BridgeFacet._handleExecuteTransaction` internal function.

However, if there is an error within `SponsorVault` contract in Optimism causing a revert when `s.sponsorVault.reimburseLiquidityFees` or `s.sponsorVault.reimburseRelayerFees` is called, the entire `execute` transaction will revert. Since `execute` transaction always revert, any cross-chain communication between Optimism and other domains will fail.

BridgeFacet.sol#L819

```
/**
 * @notice Process the transfer, and calldata if needed, when calling `execute`
 * @dev Need this to prevent stack too deep
 */
function _handleExecuteTransaction(
  ExecuteArgs calldata _args,
  uint256 _amount,
  address _asset, // adopted (or local if specified)
  bytes32 _transferId,
  bool _reconciled
) private returns (uint256) {
  // If the domain if sponsored
  if (address(s.sponsorVault) != address(0)) {
    // fast liquidity path
    if (!_reconciled) {
      // Vault will return the amount of the fee they sponsored in the native fee
      // NOTE: some considerations here around fee on transfer tokens and ensuring
      // there are no malicious `Vaults` that do not transfer the correct amount. Shoulc
      // balance read about it

      uint256 starting = IERC20(_asset).balanceOf(address(this));
      uint256 sponsored = s.sponsorVault.reimburseLiquidityFees(_asset, _args.amount, _a

      // Validate correct amounts are transferred
      if (IERC20(_asset).balanceOf(address(this)) != starting + sponsored) {
        revert BridgeFacet__handleExecuteTransaction_invalidSponsoredAmount();
      }

      _amount = _amount + sponsored;
    }

    // Should dust the recipient with the lesser of a vault-defined cap or the convertec
    // If there is no conversion available (i.e. no oracles for origin domain asset <> c
    // then the vault should just pay out the configured constant
    s.sponsorVault.reimburseRelayerFees(_args.params.originDomain, payable(_args.params.
  }
  ..SNIP..
```

Impact

It will result in denial of service. The `SponsorVault` contract, which belongs to a third-party, is a single point of failure for a domain.

Recommended Mitigation Steps

This is a problem commonly encountered whenever a method of a smart contract calls another contract – we cannot rely on the other contract to work 100% of the time, and it is dangerous to assume that the external call will always be successful. Additionally, external smart contract might be vulnerable and compromised by an attacker. Even if the team has audited or review the SponsorVault before whitelisting them, some risk might still exist.

Therefore, it is recommended to implement a fail-safe design where failure of an external call to SponsorVault will not disrupt the cross-chain communication. Consider implementing a try-catch block as shown below. If there is any issue with the external `SponsorVault` contract, no funds are reimbursed to the users in the worst case scenario, but the issue will not cause any impact to the cross-chain communication.

```
function _handleExecuteTransaction(
        ExecuteArgs calldata _args,
        uint256 _amount,
        address _asset, // adopted (or local if specified)
        bytes32 _transferId,
        bool _reconciled
) private returns (uint256) {
        // If the domain if sponsored
        if (address(s.sponsorVault) != address(0)) {
          // fast liquidity path
          if (!_reconciled) {
                // Vault will return the amount of the fee they sponsored in the native fe
                // NOTE: some considerations here around fee on transfer tokens and ensuri
                // there are no malicious `Vaults` that do not transfer the correct amount
                // balance read about it

                uint256 starting = IERC20(_asset).balanceOf(address(this));
+               try s.sponsorVault.reimburseLiquidityFees(_asset, _args.amount, _args.para
+                       // Validate correct amounts are transferred
+                       if (IERC20(_asset).balanceOf(address(this)) != starting + sponsore
+                         revert BridgeFacet__handleExecuteTransaction_invalidSponsoredAmc
+                       }
+
+                       _amount = _amount + sponsored;
+               } catch {}
          }

          // Should dust the recipient with the lesser of a vault-defined cap or the conve
          // If there is no conversion available (i.e. no oracles for origin domain asset
          // then the vault should just pay out the configured constant
+         try s.sponsorVault.reimburseRelayerFees(_args.params.originDomain, payable(_args
        ..SNIP..
```

LayneHaber (Connext) confirmed and resolved:

> Resolved by: connext/nxtp@f577ed6

0xleastwood (judge) decreased severity to Medium and commented:

> I believe this to only be a temporary DoS as the broken state can be recovered by calling `setSponsorVault`. Downgrading to `medium` risk.

[M-15] BRIDGEFACET'S _EXECUTEPORTALTRANSFER IGNORES UNDERLYING TOKEN AMOUNT WITHDRAWN FROM AAVE POOL

Submitted by hyh

`_executePortalTransfer` can introduce underlying token deficit by accounting for full underlying amount received from Aave unconditionally on what was actually withdrawn from Aave pool. Actual amount withdrawn is returned by `IAavePool(s.aavePool).withdraw()`, but currently is not used.

Setting the severity to medium as this can end up with a situation of partial insolvency, when where are a surplus of atokens, but deficit of underlying tokens in the bridge, so bridge functionality can become unavailable as there will be not enough underlying tokens, which were used up in the previous operations when atokens wasn't converted to underlying fully and underlying tokens from other operations were used up instead without accounting. I.e. the system in this situation supposes that all atokens are in the form of underlying tokens while there will be some atokens left unconverted due to withdrawal being only partial.

Proof of Concept

Call sequence here is execute() -> _handleExecuteLiquidity() -> _executePortalTransfer().

BridgeFacet._executePortalTransfer() mints the atokens needed, then withdraws them from Aave pool, always accounting for the full withdrawal:

BridgeFacet.sol#L882-L900

```
/**
 * @notice Uses Aave Portals to provide fast liquidity
 */
function _executePortalTransfer(
  bytes32 _transferId,
  uint256 _fastTransferAmount,
  address _local,
  address _router
) internal returns (uint256, address) {
  // Calculate local to adopted swap output if needed
  (uint256 userAmount, address adopted) = AssetLogic.calculateSwapFromLocalAssetIfNeeded

  IAavePool(s.aavePool).mintUnbacked(adopted, userAmount, address(this), AAVE_REFERRAL_C

  // Improvement: Instead of withdrawing to address(this), withdraw directly to the user
  IAavePool(s.aavePool).withdraw(adopted, userAmount, address(this));

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

Aave pool's withdraw() returns the amount of underlying asset that was actually withdrawn:

https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/pool/Pool.sol#L196-L217

https://github.com/aave/aave-v3-core/blob/master/contracts/protocol/libraries/logic/SupplyLogic.sol#L93-L111

If a particular lending pool has liquidity shortage at the moment, say all underlying is lent out, full withdrawal of the requested underlying token amount will not be possible.

Recommended Mitigation Steps

Consider adjusting for the amount actually withdrawn. Also the buffer that stores minted but not yet used atoken amount, say aAmountStored, can be introduced.

For example:

```
+   uint256 amountNeeded = userAmount < aAmountStored ? 0 : userAmount - aAmountStored;

-   IAavePool(s.aavePool).mintUnbacked(adopted, userAmount, address(this), AAVE_REFERRAL_C
+   if (amountNeeded > 0) {
+       IAavePool(s.aavePool).mintUnbacked(adopted, amountNeeded, address(this), AAVE_REFE
+   }

    // Improvement: Instead of withdrawing to address(this), withdraw directly to the user
-   IAavePool(s.aavePool).withdraw(adopted, userAmount, address(this));
+   uint256 amountWithdrawn = IAavePool(s.aavePool).withdraw(adopted, userAmount, address(

    // Store principle debt
-   s.portalDebt[_transferId] = userAmount;
+   s.portalDebt[_transferId] = amountWithdrawn; // can't exceed userAmount
+   aAmountStored = (userAmount < aAmountStored ? aAmountStored : userAmount) - amountWith
```

jakekidd (Connext) confirmed and commented:

> To clarify for this: We should check to make sure the full amount is withdrawn. If the full amount is not withdrawn (bc not available), revert. This way, the `execute` call can be treated in our off-chain network the same way an `execute` call using a router with insufficient funds would be treated.

jakekidd (Connext) resolved:

> Fixed by connext/nxtp@b99f9cf
>
> (Using the solution described in the above comment, not the exact code from the mitigation step above.)

0xleastwood (judge) commented:

> I agree that this improvement would provide better guarantees against insufficient unbacked funds from Aave's pools.

[M-16] DIVISION ROUNDING ERROR IN _HANDLEEXECUTELIQUIDITY() AND _RECONCILE() MAKE ROUTERBALANCES AND CONTRACT FUND BALANCE TO GET OUT OF SYNC AND CAUSE FUND LOSE

Submitted by unforgiven, also found by xiaoming90

BridgeFacet.sol#L753-L803
BridgeFacet.sol#L526-L616

Variable `routerBalances` suppose to keep track of routers balance in contract and `routers` can withdraw their balance from contract. but because of division rounding error in `_handleExecuteLiquidity()` and `_reconcile()` contract uses more of its tokens than it subtract from router's balance. this can lead to fund lose.

Proof of Concept

This is `_handleExecuteLiquidity()` code:

```
  /**
   * @notice Execute liquidity process used when calling `execute`
   * @dev Need this to prevent stack too deep
   */
  function _handleExecuteLiquidity(
    bytes32 _transferId,
    bool _isFast,
    ExecuteArgs calldata _args
  ) private returns (uint256, address) {
    uint256 toSwap = _args.amount;

    // If this is a fast liquidity path, we should handle deducting from applicable router
    // If this is a slow liquidity path, the transfer must have been reconciled (if we've 
    // and the funds would have been custodied in this contract. The exact custodied amour
    // (since the amount is hashed in the transfer ID itself) - thus, no updates are requi
    if (_isFast) {
      uint256 pathLen = _args.routers.length;

      // Calculate amount that routers will provide with the fast-liquidity fee deducted.
      toSwap = _getFastTransferAmount(_args.amount, s.LIQUIDITY_FEE_NUMERATOR, s.LIQUIDITY

      // Save the addressess of all routers providing liquidity for this transfer.
      s.routedTransfers[_transferId] = _args.routers;

      // If router does not have enough liquidity, try to use Aave Portals.
      // only one router should be responsible for taking on this credit risk, and it shou
      // deal with transfers expecting adopted assets (to avoid introducing runtime slippa
      if (
        !_args.params.receiveLocal &&
        pathLen == 1 &&
        s.routerBalances[_args.routers[0]][_args.local] < toSwap &&
        s.aavePool != address(0)
      ) {
        if (!s.routerPermissionInfo.approvedForPortalRouters[_args.routers[0]])
          revert BridgeFacet__execute_notApprovedForPortals();

        // Portal provides the adopted asset so we early return here
        return _executePortalTransfer(_transferId, toSwap, _args.local, _args.routers[0]);
      } else {
        // for each router, assert they are approved, and deduct liquidity
        uint256 routerAmount = toSwap / pathLen;
        for (uint256 i; i < pathLen; ) {
          // decrement routers liquidity
          s.routerBalances[_args.routers[i]][_args.local] -= routerAmount;

          unchecked {
            i++;
          }
        }
      }
    }
```

As you can see in last part it uses `uint256 routerAmount = toSwap / pathLen` to calculate amount to decrease from router's balance but because of division rounding error contract using `toSwap` amount of token buy total value it decrease from router's balances are lower than `toSwap`.

Of course in `_reconcile()` contract add some amount to router's balances again with division rounding error, but because the amounts are different in this two functions (in `_handleExecuteLiquidity()` we subtract fee and in `_reconcile()` we don't subtract fee) so the division rounding error would be different for them and in the end sum of total balances of routers would not be equal to contract balance. this bug could be more serious for tokens with low precision and higher price.

Tools Used

VIM

Recommended Mitigation Steps

Perform better calculations.

LayneHaber (Connext) acknowledged, but disagreed with severity and commented:

> The maximum fund loss in this case is capped by the maximum number of routers allowed in a transfer (this will generally be below 10, meaning maximum loss from one of these errors is 18 wei units of the token — assuming it hit max on both execute and reconcile).
>
> I understand the rounding error exists, but even with tokens at a precision of 6 with a high price the maximum rounding error is small (i.e. 100K coin, 6 decimals, this amounts to $1.8). At this level of impact, this should amount to a value leak.

0xleastwood (judge) decreased severity to QA and commented:

> I'm gonna downgrade this to `QA` (low risk / non-critical) because the value leak is mostly negligible and extremely hard to avoid. You could sweep the remaining balance and delegate that to the last router, but this is unfair to other routers.
>
> I think a good solution would be to make the last router pay the swept balance and then be reconciled this amount once the bridge transfer is complete.

0xleastwood (judge) increased severity to Medium and commented:

> Actually, upon thinking about this more, I think there is potential for the system to not work as intended under certain parts of the transfer flow.
>
> If the bridge transfer amount is `10 DAI` and the liquidity must be provided *three* routers, each router provides `3 DAI` respectively. If the user opted to receive the local asset than `10 DAI` will be directly sent out to them. Therefore, until the transfer has been reconciled, the protocol will essentially take on temporary bad debt which won't be resolved until the bridge transfer has been reconciled. This may limit withdrawals for other routers during this period. For these reasons, I think `medium` severity makes more sense:)

[M-17] SWAPS DONE INTERNALLY WILL BE NOT BE POSSIBLE

Submitted by 0xmint

BridgeFacet.sol#L346
BridgeFacet.sol#L812

Vulnerability details

Affected functions(that rely on swapAsset()) are:

AssetLogic.sol#L193

AssetLogic.sol#L159

swapAsset() facilitates two swaps, either using the internal or external pool. But if an internal pool exists, a swap will be unsuccessful because the call to

`s.swapStorages[_canonicalId].swapInternal()` takes two incorrect arguments (due to an incorrect ordering, this seemed to be an oversight, acknowledged by #Layne) :

AssetLogic.sol#L278-L279

Based on the above mentioned code , the arguments would be incorrectly changed to :

SwapUtils.sol#L744-L745

The condition checked here:

SwapUtils.sol#L750

will never be true as the msg.sender would never own the quantity of tokens being swapped from since it's the wrong token.

jakekidd (Connext) confirmed and resolved:

> Resolved by connext/nxtp@a66844e

0xleastwood (judge) commented:

> Great find!

[M-18] REPAYING AAVE LOAN IN _LOCAL RATHER THAN ADOPTED ASSET

Submitted by cloudjunky

When repaying the AAVE Portal in `repayAavePortal()` the `_local` asset is used to repay the loan in `_backLoan()` rather than the `adopted` asset. This is likely to cause issues in production when actually repaying loans if the asset/token being repayed to AAVE is not the same as the asset/token that was borrowed.

Proof of Concept

The comment on `L93` of `PortalFacet.sol` states;

```
// Need to swap into adopted asset or asset that was backing the loan
// The router will always be holding collateral in the local asset while the loaned asset
// is the adopted asset
```

The swap is executed on `L98` in the call to `AssetLogic.swapFromLocalAssetIfNeededForExactOut()` however the return value `adopted` is never used (it's an unused local variable). The full function is shown below;

```
// Swap for exact `totalRepayAmount` of adopted asset to repay aave
(bool success, uint256 amountIn, address adopted) = AssetLogic.swapFromLocalAssetIfNeededF
  _local,
  totalAmount,
  _maxIn
);

if (!success) revert PortalFacet__repayAavePortal_swapFailed();

// decrement router balances
unchecked {
  s.routerBalances[msg.sender][_local] -= amountIn;
}

// back loan
_backLoan(_local, _backingAmount, _feeAmount, _transferId);
```

The balance of the `_local` token is reduced but instead of the `adopted` token being passed to `_backLoan()` in L112 the `_local` token is used.

Tools Used

Vim

Recommended Mitigation Steps

To be consistent with the comments in the `repayAavePortal()` function `adopted` should be passed to `_backLoan` so that the loan is repayed in the appropriate token.

Remove the reference to `_local` in the `_backLoan()` function and replace it with `adopted` so it reads;

`_backLoan(adopted, _backingAmount, _feeAmount, _transferId);`

jakekidd (Connext) confirmed and resolved

0xleastwood (judge) commented:

> If routers are able to repay Aave debt using a token of higher denomination (i.e. pay `USDC` debt using `ETH` or `DAI`), then the liquidity portal functionality will be broken until Connext applies the necessary protocol upgrade to become solvent again.
>
> Considering the fact that only whitelisted routers have access to this feature and the bridge transfers are not broken but instead limited, I think `medium` severity makes sense.

[M-19] ATTACKER CAN PERFORM GRIEFING FOR PROCESS() IN PROMISEROUTER BY REVERTING CALLS TO CALLBACK() IN CALLBACKADDRESS

Submitted by unforgiven

PromiseRouter.sol#L226-L262

`process()` in `PromiseRouter` is used for process stored callback function and anyone calls it gets `callbackFee` and it calls `callback()` function of `callbackAddress`. but attacker set a `callbackAddress` that reverts on `callback()` and cause `process()` caller griefing. attacker can perform this buy front running or complex logic.

Proof of Concept

This is `process()` code:

```
/**
 * @notice Process stored callback function
 * @param transferId The transferId to process
 */
function process(bytes32 transferId, bytes calldata _message) public nonReentrant {
  // parse out the return data and callback address from message
  bytes32 messageHash = messageHashes[transferId];
  if (messageHash == bytes32(0)) revert PromiseRouter__process_invalidTransferId();

  bytes29 _msg = _message.ref(0).mustBePromiseCallback();
  if (messageHash != _msg.keccak()) revert PromiseRouter__process_invalidMessage();

  // enforce relayer is whitelisted by calling local connext contract
  if (!connext.approvedRelayers(msg.sender)) revert PromiseRouter__process_notApprovedRe

  address callbackAddress = _msg.callbackAddress();

  if (!AddressUpgradeable.isContract(callbackAddress)) revert PromiseRouter__process_not

  uint256 callbackFee = callbackFees[transferId];

  // remove message
  delete messageHashes[transferId];

  // remove callback fees
  callbackFees[transferId] = 0;

  // execute callback
  ICallback(callbackAddress).callback(transferId, _msg.returnSuccess(), _msg.returnData(

  emit CallbackExecuted(transferId, msg.sender);

  // Should transfer the stored relayer fee to the msg.sender
```

```
    if (callbackFee > 0) {
      AddressUpgradeable.sendValue(payable(msg.sender), callbackFee);
    }
  }
```

As you can see it calls `ICallback(callbackAddress).callback(transferId, _msg.returnSuccess(), _msg.returnData());` and if that call reverts then whole transaction would revert. so attacker can setup `callbackAddress` that revert and the caller of `process()` wouldn't get any fee and lose gas.

Tools Used

VIM

Recommended Mitigation Steps

Change the code so it won't revert if call to `callbackAddress` reverts.

jakekidd (Connext) confirmed, but disagreed with severity

jakekidd (Connext) commented:

> If the callback would revert, *normally* it won't be called.
>
> However, the attacker (griefer) could potentially frontrun the relayer's callback transaction (which is already submitted / in the mempool) and update the state of their callback contract in such a way to cause this subsequent callback to fail. Why would they do that? Nothing is gained, only losses are incurred on both sides.
>
> This seems like it should be invalid, but it also seems like we should be doing a try/catch on principle though. Perhaps the issue is misrepresented here - it's not so much an attack vector as it's 'best practice' / QA issue.
>
> ~~Let's de-escalate to QA issue.~~ Confirming, but would be great to get a second look from @LayneHaber on this.
>
> EDIT: Changed my mind, think the risk level is appropriate.

LayneHaber (Connext) changed to acknowledged and commented:

> If you change the message, the following check(s) would fail (since the hash is put onchain when the message is propagated by nomad):
>
> ```
> bytes32 messageHash = messageHashes[transferId];
> if (messageHash == bytes32(0)) revert PromiseRouter__process_invalidTr
>
> bytes29 _msg = _message.ref(0).mustBePromiseCallback();
> if (messageHash != _msg.keccak()) revert PromiseRouter__process_invali
> ```

> An attacker cannot falsify this message because sending messages on the router is restricted via the `onlyConnext` modifier, so I don't think changing the `callbackAddress` is a valid attack strategy.
>
> If we extend this concern to any failing `callback`, then any revert would not be executed. In most cases, this would be caught in offchain simulations meaning the relayer would not submit the transaction (and then nobody could process the callback). This means that integrators *must* handle failures within their code. This is a common practice when writing crosschain handlers (i.e. nomad `handle` should not revert as the message cannot be reprocessed, functions called via `execute` should handle errors unless sending funds to a recovery address is okay, etc.).
>
> Don't think this qualifies as a value leak, but because it is potentially unprocessable will leave the severity at 2 and move the label to "acknowledged".

0xleastwood (judge) commented:

> I guess this issue has the same concerns as M-01 (Issue #220). Even if the onus is on the caller of `process` to simulate the call beforehand, it seems likely that the transaction could be front-run and prove to be poor user experience for relayers.
>
> I don't think its realistic that relayers would call this function without first simulating the transaction to see if the callback fails, but I guess the bridge user could set the callback address up as a honey pot such that simulated transactions are successful.
>
> It probably makes sense to remove the main culprit of the issue by using a try/catch statement for the `ICallback(callbackAddress).callback(transferId, _msg.returnSuccess(), _msg.returnData());` call.

[M-20] IN REIMBURSELIQUIDITYFEES() OF SPONSERVAULT CONTRACT SWAPS TOKENS WITHOUT SLIPPAGE LIMIT SO ITS POSSIBLE TO PERFORM SANDWICH ATTACK AND IT CREATE MEV

Submitted by unforgiven

SponsorVault.sol#L187-L220

When code swaps tokens it should specify slippage but in `reimburseLiquidityFees()` code contract calls `tokenExchange.swapExactIn()` without slippage and it's possible to perform sandwich attack and make contract to swap on bad exchange rates and there is MEV.

Proof of Concept

This is `reimburseLiquidityFees()` code in `SponserVault`:

```
  /**
```

```
 * @notice Performs liquidity fee reimbursement.
 * @dev Uses the token exchange or liquidity deposited in this contract.
 *      The `_receiver` address is only used for emitting in the event.
 * @param _token The address of the token
 * @param _liquidityFee The liquidity fee amount
 * @param _receiver The address of the receiver
 * @return Sponsored liquidity fee amount
 */
function reimburseLiquidityFees(
  address _token,
  uint256 _liquidityFee,
  address _receiver
) external override onlyConnext returns (uint256) {
  uint256 sponsoredFee;

  if (address(tokenExchanges[_token]) != address(0)) {
    uint256 currentBalance = address(this).balance;
    ITokenExchange tokenExchange = tokenExchanges[_token];

    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);
  } else {
    uint256 balance = IERC20(_token).balanceOf(address(this));
    sponsoredFee = balance < _liquidityFee ? balance : _liquidityFee;

    // some ERC20 do not allow to transfer 0 amount
    if (sponsoredFee > 0) {
      IERC20(_token).safeTransfer(msg.sender, sponsoredFee);
    }
  }

  emit ReimburseLiquidityFees(_token, sponsoredFee, _receiver);

  return sponsoredFee;
}
```

As you can see there is no slippage defined when calling `swapExactIn()` can that swap could happen in any exchange rate. it's possible to perform sandwich attack and do large swap before and after the transaction and make users lose funds. and it's also MEV opportunity.

Tools Used

VIM

Recommended Mitigation Steps

Specify slippage when calling swap tokens.

jakekidd (Connext) acknowledged and commented:

> This is absolutely correct, but unfortunately there is no apparent viable on-chain solution. If we revert because slippage is too high, behavior would be inconsistent and UX would be bad, to say the least. If we instead choose to *not sponsor* because the slippage is too high, then sponsorship could be griefed severely (i.e. 'if we can't get at least X, then give the user nothing' is not a valid sponsorship policy).
>
> Some things that make a sandwich attack less feasible however:

> > Using exchanges with deep liquidity provision (sandwich attacks require principle liquidity, and cannot leverage flashloans).
> >
> > The fact that the liquidity fee here is only .05% of the transfer means that it would take a very heavy-handed attack and at least 1 very large transfer to make it profitable (for a 1M transfer, max profit is 500 dollars).
>
> This is a good entrypoint for a future feature that gives sponsors better options, such as only sponsoring transfers for specific assets that they supply (i.e. so no swap is required).
>
> cc @LayneHaber for thoughts. Because this essentially boils down to a feature request on behalf of sponsors - not a bug.

0xleastwood (judge) commented:

> Seeing as I can't verify if `tokenExchange.swapExactIn` is indeed vulnerable to slippage, I will just side with the sponsor/warden on this one.

Low Risk and Non-Critical Issues

For this contest, 57 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by BowTiedWardens received the top score from the judge.

The following wardens also submitted reports: 0x1f8b, cccz, unforgiven, cryptphi, xiaoming90, WatchPug, 0xf15ers, bardamu, cmichel, oyc_109, IllIllI, JMukesh, joestakey, Chom, hansfriese, SmartSek, sorrynotsorry, catchup, Jujic, kenta, Ruhum, 0xNineDec, simon135, TerrierLover, 0x52, 0xNazgul, asutorufos, ch13fd357r0y3r, k, Lambda, robee, shenwilly, SooYa, tintin, TomJ, _Adam, 0xkatana, auditor0517, cloudjunky, csanuragjain, defsec, ElKu, fatherOfBlocks, Funen, hyh, jayjonah8, sach1r0, 0x29A, 0xmint, c3phas, Kaiziron, MiloTruck, obtarian, slywaters, Waze, and zzzitron.

TABLE OF CONTENTS

result to a hash function such as `keccak256()`

[L-10] Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions

[L-11] All `initialize()` functions are front-runnable in the solution

[L-12] Use the same revert string for consistency when testing the same condition

[L-13] Use a `constant` instead of duplicating the same string

[L-14] Use a 2-step ownership transfer pattern

[L-15] A magic number should be documented and explained. Use a `constant` instead

[L-16] Lack of event emission for operation changing the state

Non-Critical Issues

[N-01] It's better to emit after all processing is done

[N-02] The `nonReentrant` `modifier` should occur before all other modifiers

[N-03] Typos

[N-04] Deprecated library used for Solidity `>= 0.8` : SafeMath

[N-05] Open TODOS

[N-06] Adding a `return` statement when the function defines a named return variable, is redundant

[N-07] The pragmas used are not the same everywhere

[N-08] Non-library/interface files should use fixed compiler versions, not floating ones

[N-09] Missing NatSpec

[L-01] `AL`, `FEE` AND `ADMINFEE` CANNOT BE SET THE THEIR MAXIMUM VALUE

Consider replacing `<` with `<=` here:

```
File: StableSwap.sol
96:     require(_a < AmplificationUtils.MAX_A, "_a exceeds maximum");
```

[L-02] ADD CONSTRUCTOR INITIALIZERS

As per OpenZeppelin's (OZ) recommendation, "The guidelines are now to make it impossible for *anyone* to run `initialize` on an implementation contract, by adding an empty constructor with the `initializer` modifier. So the implementation contract gets initialized automatically upon deployment."

Note that this behaviour is also incorporated the OZ Wizard since the UUPS vulnerability discovery: “Additionally, we modified the code generated by the Wizard 19 to include a constructor that automatically initializes the implementation when deployed.”

Furthermore, this thwarts any attempts to frontrun the initialization tx of these contracts:

```
core/connext/helpers/BridgeToken.sol:34:  function initialize() public override initialize
core/connext/helpers/LPToken.sol:21:  function initialize(string memory name, string memor
core/connext/helpers/StableSwap.sol:61:  function initialize(
core/connext/helpers/TokenRegistry.sol:73:  function initialize(address _tokenBeacon, addr
core/connext/interfaces/IBridgeToken.sol:5:  function initialize() external;
core/connext/interfaces/IStableSwap.sol:99:  function initialize(
core/promise/PromiseRouter.sol:146:  function initialize(address _xAppConnectionManager) p
core/relayer-fee/RelayerFeeRouter.sol:80:  function initialize(address _xAppConnectionMana
```

[L-03] UNSAFE CASTING MAY OVERFLOW

SafeMath and Solidity 0.8.* handles overflows for basic math operations but not for casting. Consider using OpenZeppelin’s SafeCast library to prevent unexpected overflows when casting from uint256 here:

```
core/connext/libraries/ConnextMessage.sol:49:    _view.assertType(uint40(_t));
core/connext/libraries/ConnextMessage.sol:71:    return actionType(_action) == uint8(_type
core/connext/libraries/ConnextMessage.sol:138:      abi.encodePacked(Types.Transfer, _to,
core/connext/libraries/ConnextMessage.sol:157:    return abi.encodePacked(_domain, _id).re
core/connext/libraries/ConnextMessage.sol:188:      return _message.castTo(uint40(Types.Me
core/connext/libraries/ConnextMessage.sol:201:    return Types(uint8(_view.typeOf()));
core/connext/libraries/ConnextMessage.sol:210:    return _message.slice(0, TOKEN_ID_LEN, u
core/connext/libraries/ConnextMessage.sol:220:    uint40 _type = uint40(msgType(_message))
core/connext/libraries/ConnextMessage.sol:232:    return uint32(_tokenId.indexUint(0, 4));
core/connext/libraries/ConnextMessage.sol:263:    return uint8(_message.indexUint(TOKEN_ID
core/connext/libraries/ConnextMessage.sol:272:    return uint8(_action.indexUint(0, 1));
core/connext/libraries/Encoding.sol:37:      uint8 _b = uint8(_bytes >> (i * 8));
core/connext/libraries/Encoding.sol:46:        uint8 _b = uint8(_bytes >> (i * 8));
core/connext/libraries/Encoding.sol:62:    _char = uint8(NIBBLE_LOOKUP[_nibble]);
core/connext/libraries/LibCrossDomainProperty.sol:48:    _view.assertType(uint40(_t));
core/connext/libraries/LibCrossDomainProperty.sol:71:    return propertyType(_property) ==
core/connext/libraries/LibCrossDomainProperty.sol:89:    return uint8(_property.indexUint(
core/connext/libraries/LibCrossDomainProperty.sol:99:      return _view.castTo(uint40(Type
core/connext/libraries/LibCrossDomainProperty.sol:130:    return uint32(_property.indexUin
core/connext/libraries/LibCrossDomainProperty.sol:140:    return abi.encodePacked(Types.Dc
core/connext/libraries/LibDiamond.sol:124:    uint96 selectorPosition = uint96(ds.facetFur
core/connext/libraries/LibDiamond.sol:142:    uint96 selectorPosition = uint96(ds.facetFur
core/connext/libraries/LibDiamond.sol:201:      ds.selectorToFacetAndPosition[lastSelector
core/promise/libraries/PromiseMessage.sol:41:    _view.assertType(uint40(_t));
core/promise/libraries/PromiseMessage.sol:63:        uint8(Types.PromiseCallback),
core/promise/libraries/PromiseMessage.sol:66:        uint8(_returnSuccess ? 1 : 0),
core/promise/libraries/PromiseMessage.sol:152:      return _view.castTo(uint40(Types.Promi
core/promise/PromiseRouter.sol:313:    return (uint64(_origin) << 32) | _nonce;
core/relayer-fee/libraries/RelayerFeeMessage.sol:44:    _view.assertType(uint40(_t));
core/relayer-fee/libraries/RelayerFeeMessage.sol:57:    return abi.encodePacked(uint8(Type
core/relayer-fee/libraries/RelayerFeeMessage.sol:109:      return _view.castTo(uint40(Type
core/relayer-fee/RelayerFeeRouter.sol:166:    return (uint64(_origin) << 32) | _nonce;
```

[L-04] UNINITIALIZED UPGRADEABLE CONTRACT

Similar issue in the past: here

Upgradeable dependencies should be initialized.
While not causing any harm at the moment, suppose OZ someday decide to upgrade this contract and the sponsor uses the new version: it is possible that the contract will not work anymore.
Consider calling the `init()` here:

```
File: PromiseRouter.sol
23: contract PromiseRouter is Version, Router, ReentrancyGuardUpgradeable {
...
146:   function initialize(address _xAppConnectionManager) public initializer {
147:     __XAppConnectionClient_initialize(_xAppConnectionManager);
+ 147:     __ReentrancyGuard_init(); //@audit ReentrancyGuardUpgradeable
148:   }
```

This is already applied L72 in `StableSwap.sol` (but was forgotten in `PromiseRouter.sol`):

```
File: StableSwap.sol
61:   function initialize(
...
70:   ) public override initializer {
71:     __OwnerPausable_init();
72:     __ReentrancyGuard_init();
```

[L-05] DEPRECATED SAFEAPPROVE() FUNCTION

Deprecated

Using this deprecated function can lead to unintended reverts and potentially the locking of funds. A deeper discussion on the deprecation of this function is in OZ issue #2219 (OpenZeppelin/openzeppelin-contracts#2219). The OpenZeppelin ERC20 `safeApprove()` function has been deprecated, as seen in the comments of the OpenZeppelin code.

As recommended by the OpenZeppelin comment, consider replacing `safeApprove()` with `safeIncreaseAllowance()` or `safeDecreaseAllowance()` instead:

```
core/connext/libraries/AssetLogic.sol:347:       SafeERC20.safeApprove(IERC20(_assetIn),
```

[L-06] MISSING ADDRESS(0) CHECKS

Consider adding an `address(0)` check for immutable variables:

File: Executor.sol

```
29:  address private immutable connext;
...
47:   constructor(address _connext) {
+ 48:     require(_connext != address(0));
48:     connext = _connext;
49:   }
```

[L-07] MISLEADING COMMENT

Issue with comment

I suspect a copy-paste error here:

```
- core/connext/libraries/SwapUtils.sol:790:    require(dx <= maxDx, "Swap didn't result ir
+ core/connext/libraries/SwapUtils.sol:790:    require(dx <= maxDx, "Swap needs more than
```

[L-08] ADD A TIMELOCK TO CRITICAL FUNCTIONS

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate (less risk of a malicious owner making a sandwich attack on a user).

Consider adding a timelock to:

```
core/connext/facets/StableSwapFacet.sol:469:  function setSwapAdminFee(bytes32 canonicalIc
core/connext/facets/StableSwapFacet.sol:478:  function setSwapFee(bytes32 canonicalId, uir
core/connext/helpers/StableSwap.sol:448:  function setAdminFee(uint256 newAdminFee) exterr
core/connext/helpers/StableSwap.sol:456:  function setSwapFee(uint256 newSwapFee) external
```

[L-09] ABI.ENCODEPACKED() SHOULD NOT BE USED WITH DYNAMIC TYPES WHEN PASSING THE RESULT TO A HASH FUNCTION SUCH AS KECCAK256()

Similar issue in the past: here

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead.

```
core/connext/helpers/BridgeToken.sol:134:    bytes32 _digest = keccak256(abi.encodePacked(
core/connext/libraries/ConnextMessage.sol:178:    return keccak256(abi.encodePacked(bytes(
```

[L-10] UPGRADEABLE CONTRACT IS MISSING A __GAP[50] STORAGE VARIABLE TO ALLOW FOR NEW STORAGE VARIABLES IN LATER VERSIONS

See this link for a description of this storage variable. While some contracts may not currently be sub-classed, adding the variable now protects against forgetting to add it in the future:

- LPToken.sol
- OwnerPausableUpgradeable.sol

Those contracts do apply the recommendation though:

- ProposedOwnableUpgradeable.sol
- BridgeToken.sol
- ProposedOwnable.sol
- Router.sol
- XAppConnectionClient.sol

[L-11] ALL INITIALIZE() FUNCTIONS ARE FRONT-RUNNABLE IN THE SOLUTION

Consider adding some access control to them or deploying atomically or using constructor `initializer`:

```
core/connext/helpers/BridgeToken.sol:34:  function initialize() public override initialize
core/connext/helpers/LPToken.sol:21:  function initialize(string memory name, string memor
core/connext/helpers/StableSwap.sol:61:  function initialize(
core/connext/helpers/TokenRegistry.sol:73:  function initialize(address _tokenBeacon, addr
core/connext/interfaces/IBridgeToken.sol:5:  function initialize() external;
core/connext/interfaces/IStableSwap.sol:99:  function initialize(
core/promise/PromiseRouter.sol:146:  function initialize(address _xAppConnectionManager) p
core/relayer-fee/RelayerFeeRouter.sol:80:  function initialize(address _xAppConnectionMana
```

[L-12] USE THE SAME REVERT STRING FOR CONSISTENCY WHEN TESTING THE SAME CONDITION

Issue with comment

Consider only using the shorter string:

```
core/connext/helpers/StableSwap.sol:155:    require(index < swapStorage.pooledTokens.lengt
core/connext/helpers/StableSwap.sol:177:    require(index < swapStorage.pooledTokens.lengt
```

[L-13] USE A CONSTANT INSTEAD OF DUPLICATING THE SAME STRING

Issue with comment

```
core/connext/libraries/LibDiamond.sol:123:    require(_facetAddress != address(0), "LibDia
core/connext/libraries/LibDiamond.sol:141:    require(_facetAddress != address(0), "LibDia
```

```
core/connext/libraries/LibDiamond.sol:121:    require(_functionSelectors.length > 0, "LibD
core/connext/libraries/LibDiamond.sol:139:    require(_functionSelectors.length > 0, "LibD
core/connext/libraries/LibDiamond.sol:158:    require(_functionSelectors.length > 0, "LibD
```

```
core/connext/libraries/SwapUtils.sol:493:    require(tokenIndexFrom < xp.length && tokenIn
core/connext/libraries/SwapUtils.sol:524:    require(tokenIndexFrom < xp.length && tokenIn
```

```
core/connext/libraries/SwapUtils.sol:662:    require(dy >= minDy, "Swap didn't result in m
core/connext/libraries/SwapUtils.sol:756:    require(dy >= minDy, "Swap didn't result in m
```

```
core/connext/libraries/SwapUtils.sol:703:    require(dx <= maxDx, "Swap needs more than ma
core/connext/libraries/SwapUtils.sol:790:    require(dx <= maxDx, "Swap didn't result in m
```

```
core/connext/libraries/SwapUtils.sol:697:    require(dy <= self.balances[tokenIndexTo], "C
core/connext/libraries/SwapUtils.sol:784:    require(dy <= self.balances[tokenIndexTo], "C
```

```
core/connext/libraries/SwapUtils.sol:717:      require(dx <= tokenFrom.balanceOf(msg.sende
core/connext/libraries/SwapUtils.sol:649:      require(dx <= tokenFrom.balanceOf(msg.sende
core/connext/libraries/SwapUtils.sol:750:    require(dx <= tokenFrom.balanceOf(msg.sender)
```

```
core/connext/libraries/SwapUtils.sol:1071:    require(newAdminFee <= MAX_ADMIN_FEE, "Fee i
core/connext/libraries/SwapUtils.sol:1084:    require(newSwapFee <= MAX_SWAP_FEE, "Fee is
```

[L-14] USE A 2-STEP OWNERSHIP TRANSFER PATTERN

Contracts inheriting from OpenZeppelin's libraries have the default `transferOwnership()` function (a one-step process). It's possible that the `onlyOwner` role mistakenly transfers ownership to a wrong address, resulting in a loss of the `onlyOwner` role. Consider overriding the default `transferOwnership()` function to first nominate an address as the `pendingOwner` and implementing an `acceptOwnership()` function which is called by the `pendingOwner` to confirm the transfer.

```
core/connext/helpers/BridgeToken.sol:202:  function transferOwnership(address _newOwner) p
core/connext/helpers/BridgeToken.sol:203:    OwnableUpgradeable.transferOwnership(_newOwne
core/connext/helpers/TokenRegistry.sol:302:    IBridgeToken(_token).transferOwnership(owne
core/connext/interfaces/IBridgeToken.sol:29:  // inherited from ownable
core/connext/interfaces/IBridgeToken.sol:30:  function transferOwnership(address _newOwner
```

[L-15] A MAGIC NUMBER SHOULD BE DOCUMENTED AND EXPLAINED. USE A `CONSTANT` INSTEAD

Similar issue in the past: here

```
core/connext/facets/upgrade-initializers/DiamondInit.sol:70:      s.nonce = 0;
core/connext/facets/upgrade-initializers/DiamondInit.sol:76:      s.LIQUIDITY_FEE_NUMERATC
core/connext/facets/upgrade-initializers/DiamondInit.sol:77:      s.LIQUIDITY_FEE_DENOMINA
core/connext/facets/upgrade-initializers/DiamondInit.sol:78:      s.maxRoutersPerTransfer
core/connext/helpers/TokenRegistry.sol:300:    IBridgeToken(_token).setDetails(_name, _syn
```

Consider using `constant` variables as this would make the code more maintainable and readable while costing nothing gas-wise (constants are replaced by their value at compile-time).

[L-16] LACK OF EVENT EMISSION FOR OPERATION CHANGING THE STATE

```
File: AssetFacet.sol
171:   function removeAssetId(bytes32 _canonicalId, address _adoptedAssetId) external only
...
179:     delete s.adoptedToLocalPools[_canonicalId]; // @audit-info INFO no event emitted
```

[N-01] IT'S BETTER TO EMIT AFTER ALL PROCESSING IS DONE

contracts/contracts/core/connext/facets/AssetFacet.sol:

```
  152:     // Emit event
  153:     emit AssetAdded(_canonical.id, _canonical.domain, _adoptedAssetId, supported, m
  154
  155      // Add the swap pool
  156      _addStableSwapPool(_canonical, _stableSwapPool);
  157    }
```

contracts/contracts/core/connext/facets/RoutersFacet.sol:

```
  303:     // Emit event
  304:     emit RouterRemoved(router, msg.sender);
  305
  306      // Remove router owner
  307      address _owner = s.routerPermissionInfo.routerOwners[router];
  308      if (_owner != address(0)) {
  309:       emit RouterOwnerAccepted(router, _owner, address(0));
  310        // delete routerOwners[router];
  311        s.routerPermissionInfo.routerOwners[router] = address(0);
  312      }



  316      if (_recipient != address(0)) {
  317:       emit RouterRecipientSet(router, _recipient, address(0));
  318        // delete routerRecipients[router];
  319        s.routerPermissionInfo.routerRecipients[router] = address(0);
  320      }
```

```
335:     emit MaxRoutersPerTransferUpdated(_newMaxRouters, msg.sender);
336
337      s.maxRoutersPerTransfer = _newMaxRouters;
338    }
```

contracts/contracts/core/connext/helpers/ConnextPriceOracle.sol:

```
158    function setDirectPrice(address _token, uint256 _price) external onlyAdmin {
159:     emit DirectPriceUpdated(_token, assetPrices[_token], _price);
160      assetPrices[_token] = _price;
161    }
```

```
163    function setV1PriceOracle(address _v1PriceOracle) external onlyAdmin {
164:     emit V1PriceOracleUpdated(v1PriceOracle, _v1PriceOracle);
165      v1PriceOracle = _v1PriceOracle;
166    }
```

contracts/contracts/core/connext/helpers/SponsorVault.sol:

```
150:     emit RateUpdated(_originDomain, rates[_originDomain], _rate, msg.sender);
151
152      rates[_originDomain] = _rate;
153    }
```

```
159    function setRelayerFeeCap(uint256 _relayerFeeCap) external onlyOwner {
160:     emit RelayerFeeCapUpdated(relayerFeeCap, _relayerFeeCap, msg.sender);
161      relayerFeeCap = _relayerFeeCap;
162    }
```

```
168    function setGasTokenOracle(address _gasTokenOracle) external onlyOwner {
169:     emit GasTokenOracleUpdated(address(gasTokenOracle), _gasTokenOracle, msg.sender
170      gasTokenOracle = IGasTokenOracle(_gasTokenOracle);
171    }
```

```
181:     emit TokenExchangeUpdated(_token, address(tokenExchanges[_token]), _tokenExchar
182      tokenExchanges[_token] = ITokenExchange(_tokenExchange);
183    }
```

contracts/contracts/core/connext/libraries/LibDiamond.sol:

```
    emit DiamondCut(_diamondCut, _init, _calldata);
    initializeDiamondCut(_init, _calldata);
  }
```

contracts/contracts/core/promise/PromiseRouter.sol:

```
    emit CallbackExecuted(transferId, msg.sender);

    // Should transfer the stored relayer fee to the msg.sender
    if (callbackFee > 0) {
      AddressUpgradeable.sendValue(payable(msg.sender), callbackFee);
```

[N-02] THE NONREENTRANT MODIFIER SHOULD OCCUR BEFORE ALL OTHER MODIFIERS

This is a best-practice to protect against re-entrancy in other modifiers

```
core/connext/facets/BridgeFacet.sol:279:  function xcall(XCallArgs calldata _args) externa
core/connext/facets/BridgeFacet.sol:411:  function execute(ExecuteArgs calldata _args) ext
```

[N-03] TYPOS

funciton

```
core/connext/facets/upgrade-initializers/DiamondInit.sol:31:// of your diamond. Add parame
```

_potentialReplcia

```
core/connext/facets/BaseConnextFacet.sol:137:   * @notice Determine whether _potentialRepl
```

bridgable

```
core/connext/facets/BridgeFacet.sol:265:   * assets will be swapped for their local nomad
```

addressess

```
core/connext/facets/BridgeFacet.sol:774:      // Save the addressess of all routers provic
```

a sthe

```
core/connext/facets/BridgeFacet.sol:1059:    // Because we are using the `_amount` a sthe
```

sournce

```
core/connext/facets/ProposedOwnableFacet.sol:143:    // Contract as sournce of truth
core/connext/facets/ProposedOwnableFacet.sol:163:    // Contract as sournce of truth
core/connext/facets/ProposedOwnableFacet.sol:176:    // Contract as sournce of truth
core/connext/helpers/ProposedOwnableUpgradeable.sol:170:    // Contract as sournce of trut
core/connext/helpers/ProposedOwnableUpgradeable.sol:198:    // Contract as sournce of trut
```

rlayer

```
core/connext/facets/RelayerFacet.sol:35:   * @notice Emitted when a rlayer is added or rem
```

specicfied

```
core/connext/facets/RoutersFacet.sol:575:    // transfer to specicfied recipient IF recipi
```

callabale

```
core/connext/helpers/Executor.sol:17: * @notice This library contains an `execute` functic
```

everytime

```
core/connext/helpers/LPToken.sol:41:   * minting and burning. This ensures that Swap.updat
```

stabelswap

```
core/connext/libraries/AssetLogic.sol:30:   * @notice Check if the stabelswap pool exists
```

briding

```
core/connext/libraries/LibConnextStorage.sol:278:   * @notice Stores whether or not bridir
```

identifer

```
core/connext/libraries/LibCrossDomainProperty.sol:35:  uint256 private constant PROPERTY_L
```

incomming

```
core/promise/PromiseRouter.sol:50:   * @dev While handling the message, it will parse trar
```

[N-04] DEPRECATED LIBRARY USED FOR SOLIDITY >= 0.8 : SAFEMATH

```
core/connext/helpers/ConnextPriceOracle.sol:2:pragma solidity 0.8.14;
core/connext/helpers/ConnextPriceOracle.sol:4:import {SafeMath} from "@openzeppelin/contra
core/connext/helpers/ConnextPriceOracle.sol:45:  using SafeMath for uint256;

core/connext/helpers/OZERC20.sol:2:pragma solidity 0.8.14;
```

```
core/connext/helpers/OZERC20.sol:10:import "@openzeppelin/contracts/utils/math/SafeMath.sc
core/connext/helpers/OZERC20.sol:37:  using SafeMath for uint256;

core/connext/libraries/AmplificationUtils.sol:2:pragma solidity 0.8.14;
core/connext/libraries/AmplificationUtils.sol:5:import {SafeMath} from "@openzeppelin/cont
core/connext/libraries/AmplificationUtils.sol:15:  using SafeMath for uint256;

core/connext/libraries/SwapUtils.sol:2:pragma solidity 0.8.14;
core/connext/libraries/SwapUtils.sol:4:import {SafeMath} from "@openzeppelin/contracts/uti
core/connext/libraries/SwapUtils.sol:20:  using SafeMath for uint256;
```

[N-05] OPEN TODOS

Consider resolving the TODOs before deploying.

```
core/connext/facets/AssetFacet.sol:66:  function canonicalToAdopted(bytes32 _canonicalId)
core/connext/facets/AssetFacet.sol:67:    return s.canonicalToAdopted[_canonicalId];
core/connext/facets/AssetFacet.sol:150:    s.canonicalToAdopted[_canonical.id] = supportec
core/connext/facets/AssetFacet.sol:185:    delete s.canonicalToAdopted[_canonicalId];
core/connext/facets/BridgeFacet.sol:492:      // TODO: do we want to store a mapping of cu
core/connext/facets/BridgeFacet.sol:579:      // TODO: do we need to keep this
core/connext/facets/BridgeFacet.sol:1027:    // TODO: Should we call approve(0) and approv
core/connext/helpers/Executor.sol:7:// TODO: see note in below file re: npm
core/connext/interfaces/IConnextHandler.sol:22:  function canonicalToAdopted(bytes32 _canc
core/connext/libraries/AssetLogic.sol:204:    address adopted = s.canonicalToAdopted[id];
core/connext/libraries/AssetLogic.sol:244:    address adopted = s.canonicalToAdopted[id];
core/connext/libraries/AssetLogic.sol:376:    address adopted = s.canonicalToAdopted[id];
core/connext/libraries/LibConnextStorage.sol:178:  mapping(bytes32 => address) canonicalTc
core/connext/libraries/LibConnextStorage.sol:303:  // BridgeFacet (cont.) TODO: can we mov
```

[N-06] ADDING A RETURN STATEMENT WHEN THE FUNCTION DEFINES A NAMED RETURN VARIABLE, IS REDUNDANT

While not consuming more gas with the Optimizer enabled: using both named returns and a return statement isn't necessary. Removing one of those can improve code clarity.

Affected code:

- contracts/contracts/core/connext/facets/StableSwapFacet.sol:

```
212:   ) external view returns (uint256 availableTokenAmount) {
213:     return s.swapStorages[canonicalId].calculateWithdrawOneToken(tokenAmount, toker
```

- contracts/contracts/core/connext/helpers/StableSwap.sol:

```
295:     returns (uint256 availableTokenAmount)
297:     return swapStorage.calculateWithdrawOneToken(tokenAmount, tokenIndex);
```

[N-07] THE PRAGMAS USED ARE NOT THE SAME EVERYWHERE

Consider using only 1 version. Here's an example of the different pragmas:

```
core/connext/facets/upgrade-initializers/DiamondInit.sol:2:pragma solidity ^0.8.0;
core/connext/facets/AssetFacet.sol:2:pragma solidity 0.8.14;
core/connext/interfaces/IAavePool.sol:2:pragma solidity ^0.8.11;
core/shared/Router.sol:2:pragma solidity >=0.6.11;
```

[N-08] NON-LIBRARY/INTERFACE FILES SHOULD USE FIXED COMPILER VERSIONS, NOT FLOATING ONES

```
core/connext/facets/upgrade-initializers/DiamondInit.sol:2:pragma solidity ^0.8.0;
core/shared/Router.sol:2:pragma solidity >=0.6.11;
core/shared/XAppConnectionClient.sol:2:pragma solidity >=0.6.11;
```

[N-09] MISSING NATSPEC

```
File: AssetFacet.sol
121:   /**
122:    * @notice Used to add supported assets. This is an admin only function
123:    * @dev When whitelisting the canonical asset, all representational assets would be
124:    * whitelisted as well. In the event you have a different adopted asset (i.e. PoS U
125:    * on polygon), you should *not* whitelist the adopted asset. The stable swap pool
126:    * address used should allow you to swap between the local <> adopted asset
127:    * @param _canonical - The canonical asset to add by id and domain. All representat
128:    * will be whitelisted as well
129:    * @param _adoptedAssetId - The used asset id for this domain (i.e. PoS USDC for
130:    * polygon)
131:    */
132:   function setupAsset(
133:     ConnextMessage.TokenId calldata _canonical,
134:     address _adoptedAssetId,
135:     address _stableSwapPool // @audit-info [INFO] NatSpec missing for _stableSwapPool
136:   ) external onlyOwner {
```

jakekidd (Connext) commented:

> Great linking, great format!
>
> All of these seem non-critical except for L-14, which is acknowledged and L-02/L-11, which are valid.

0xleastwood (judge) commented:

> I would tend to agree that all of these are valid.

Gas Optimizations

For this contest, 44 reports were submitted by wardens detailing gas optimizations. The report highlighted below by IllIllI received the top score from the judge.

The following wardens also submitted reports: 0xKitsune, 0xkatana, BowTiedWardens, defsec, joestakey, MiloTruck, 0x29A, Metatron, Tomio, robee, simon135, UnusualTurtle, Waze, _Adam, 0xf15ers, c3phas, hansfriese, oyc_109, slywaters, TomJ, 0x1f8b, apostle0x01, catchup, ElKu, kaden, Kaiziron, rfa, 0xNazgul, fatherOfBlocks, Fitraldys, Lambda, sach1r0, SmartSek, asutorufos, Funen, hyh, ignacio, nahnah, Randyyy, Ruhum, 0xmint, k, and csanuragjain.

SUMMARY

	Issue	Instances
G-01	Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate	1
G-02	State variables only set in the constructor should be declared `immutable`	1
G-03	Structs can be packed into fewer storage slots	2
G-04	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	10
G-05	Using `storage` instead of `memory` for structs/arrays saves gas	2
G-06	State variables should be cached in stack variables rather than re-reading them from storage	13
G-07	Multiple accesses of a mapping/array should use a local variable cache	13
G-08	`internal` functions only called once can be inlined to save gas	9
G-09	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if`-statement	4
G-10	`<array>.length` should not be looked up in every loop of a `for`-loop	19
G-11	`++i`/`i++` should be `unchecked{++i}`/`unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for`- and `while`-loops	26
G-12	`require()`/`revert()` strings longer than 32 bytes cost extra gas	17
G-13	Optimize names to save gas	3
G-14	Using `bool`s for storage incurs overhead	3
G-15	Use a more recent version of solidity	1
G-16	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	2
G-17	`>=` costs less gas than `>`	2
G-18	It costs more gas to initialize non-`constant`/non-`immutable` variables to zero than to let the default of zero be applied	22
G-19	`internal` functions not called by the contract should be removed to save deployment gas	2
G-20	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i`/`i--` too)	26
G-21	Splitting `require()` statements that use `&&` saves gas	6

G-22	Usage of `uints`/`ints` smaller than 32 bytes (256 bits) incurs overhead	142
G-23	Using `private` rather than `public` for constants, saves gas	6
G-24	Don't use `SafeMath` once the solidity version is 0.8.0 or greater	3
G-25	Duplicated `require()`/`revert()` checks should be refactored to a modifier or function	7
G-26	Multiple `if`-statements with mutually-exclusive conditions should be changed to `if`-`else` statements	1
G-27	`require()` or `revert()` statements that check input arguments should be at the top of the function	6
G-28	Empty blocks should be removed or emit something	1
G-29	Superfluous event fields	4
G-30	Use custom errors rather than `revert()`/`require()` strings to save gas	79
G-31	Functions guaranteed to revert when called by normal users can be marked `payable`	85

Total: 518 instances over 31 issues

[G-01] MULTIPLE `ADDRESS` MAPPINGS CAN BE COMBINED INTO A SINGLE `MAPPING` OF AN `ADDRESS` TO A `STRUCT`, WHERE APPROPRIATE

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

There is 1 instance of this issue:

```
File: contracts/contracts/core/connext/helpers/ConnextPriceOracle.sol   #1

62      mapping(address => PriceInfo) public priceRecords;
63:     mapping(address => uint256) public assetPrices;
```

https://github.com/code-423n4/2022-06-connext/blob/4dd6149748b635f95460d4c3924c7e3fb6716967/contracts/contracts/core/connext/helpers/ConnextPrice
L63

[G-02] STATE VARIABLES ONLY SET IN THE CONSTRUCTOR SHOULD BE DECLARED `IMMUTABLE`

Avoids a Gsset (20000 gas) in the constructor, and replaces each Gwarmacces (100 gas) with a `PUSH32` (3 gas).

There is 1 instance of this issue:

```
File: contracts/contracts/core/connext/helpers/ConnextPriceOracle.sol   #1

49:     address public wrapped;
```

https://github.com/code-423n4/2022-06-connext/blob/4dd6149748b635f95460d4c3924c7e3fb6716967/contracts/contracts/core/connext/helpers/ConnextPrice

[G-03] STRUCTS CAN BE PACKED INTO FEWER STORAGE SLOTS

Each slot saved can avoid an extra Gsset (20000 gas) for the first setting of the struct. Subsequent reads as well as writes have smaller gas savings

There are 2 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, please see the warden's full report.)

[G-04] USING CALLDATA INSTEAD OF MEMORY FOR READ-ONLY ARGUMENTS IN EXTERNAL FUNCTIONS SAVES GAS

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. Each iteration of this for-loop costs at least 60 gas (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external` function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one

There are 10 instances of this issue.

[G-05] USING STORAGE INSTEAD OF MEMORY FOR STRUCTS/ARRAYS SAVES GAS

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (2100 gas) for *each* field of the struct/array. If the fields are read from the new memory variable, they incur an additional `MLOAD` rather than a cheap stack read. Instead of declearing the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a function that requires `memory`, or if the array/struct is being read from another `memory` array/struct

There are 2 instances of this issue.

[G-06] STATE VARIABLES SHOULD BE CACHED IN STACK VARIABLES RATHER THAN RE-READING THEM FROM STORAGE

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (100 gas) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 13 instances of this issue.

[G-07] MULTIPLE ACCESSES OF A MAPPING/ARRAY SHOULD USE A LOCAL VARIABLE CACHE

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` or `calldata` variable when the value is accessed multiple times, saves ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata

There are 13 instances of this issue.

[G-08] `INTERNAL` FUNCTIONS ONLY CALLED ONCE CAN BE INLINED TO SAVE GAS

Not inlining costs 20 to 40 gas because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 9 instances of this issue.

[G-09] ADD `UNCHECKED {}` FOR SUBTRACTIONS WHERE THE OPERANDS CANNOT UNDERFLOW BECAUSE OF A PREVIOUS `REQUIRE()` OR `IF`-STATEMENT

`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a }`

There are 4 instances of this issue.

[G-10] `<ARRAY>.LENGTH` SHOULD NOT BE LOOKED UP IN EVERY LOOP OF A `FOR`-LOOP

The overheads outlined below are *PER LOOP*, excluding the first loop

storage arrays incur a Gwarmaccess (100 gas) memory arrays use `MLOAD` (3 gas)

calldata arrays use `CALLDATALOAD` (3 gas)

Caching the length changes each of these to a `DUP<N>` (3 gas), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 19 instances of this issue.

[G-11] ++I/I++ SHOULD BE UNCHECKED{++I}/UNCHECKED{I++} WHEN IT IS NOT POSSIBLE FOR THEM TO OVERFLOW, AS IS THE CASE WHEN USED IN FOR- AND WHILE-LOOPS

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves 30-40 gas per loop

There are 26 instances of this issue.

[G-12] REQUIRE()/REVERT() STRINGS LONGER THAN 32 BYTES COST EXTRA GAS

Each extra memory word of bytes past the original 32 incurs an MSTORE which costs 3 gas

There are 17 instances of this issue.

[G-13] OPTIMIZE NAMES TO SAVE GAS

`public`/`external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save 128 gas each during deployment, and renaming functions to have lower method IDs will save 22 gas per call, per sorted position shifted

There are 3 instances of this issue.

[G-14] USING BOOLS FOR STORAGE INCURS OVERHEAD

```
// Booleans are more expensive than uint256 or any type that takes up a full
// word because each write operation emits an extra SLOAD to first read the
// slot's contents, replace the bits taken up by the boolean, and then write
// back. This is the compiler's defense against contract upgrades and
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27 Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (20000 gas) when changing from 'false' to 'true', after having been 'true' in the past

There are 3 instances of this issue.

[G-15] USE A MORE RECENT VERSION OF SOLIDITY

Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There is 1 instance of this issue:

```
File: contracts/contracts/core/connext/facets/upgrade-initializers/DiamondInit.sol   #1

2:    pragma solidity ^0.8.0;
```

https://github.com/code-423n4/2022-06-connext/blob/4dd6149748b635f95460d4c3924c7e3fb6716967/contracts/contracts/core/connext/facets/upgrade-initializers/DiamondInit.sol#L2

[G-16] USING `> 0` COSTS MORE GAS THAN `!= 0` WHEN USED ON A `UINT` IN A `REQUIRE()` STATEMENT

This change saves 6 gas per instance. The optimization works until solidity version 0.8.13 where there is a regression in gas costs.

There are 2 instances of this issue.

[G-17] `>=` COSTS LESS GAS THAN `>`

The compiler uses opcodes `GT` and `ISZERO` for solidity code that uses `>`, but only requires `LT` for `>=`, which saves 3 gas

There are 2 instances of this issue.

[G-18] IT COSTS MORE GAS TO INITIALIZE NON-CONSTANT/NON-IMMUTABLE VARIABLES TO ZERO THAN TO LET THE DEFAULT OF ZERO BE APPLIED

Not overwriting the default for stack variables saves 8 gas. Storage and memory variables have larger savings

There are 22 instances of this issue.

[G-19] INTERNAL FUNCTIONS NOT CALLED BY THE CONTRACT SHOULD BE REMOVED TO SAVE DEPLOYMENT GAS

If the functions are required by an interface, the contract should inherit from that interface and use the `override` keyword

There are 2 instances of this issue.

[G-20] ++I COSTS LESS GAS THAN I++, ESPECIALLY WHEN IT'S USED IN FOR-LOOPS (--I/I-- TOO)

Saves 6 gas per loop

There are 26 instances of this issue.

[G-21] SPLITTING REQUIRE() STATEMENTS THAT USE && SAVES GAS

See this issue which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper

There are 6 instances of this issue.

[G-22] USAGE OF UINTS/INTS SMALLER THAN 32 BYTES (256 BITS) INCURS OVERHEAD

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

There are 142 instances of this issue.

[G-23] USING PRIVATE RATHER THAN PUBLIC FOR CONSTANTS, SAVES GAS

If needed, the value can be read from the verified contract source code. Saves 3406-3606 gas in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table

There are 6 instances of this issue.

[G-24] DON'T USE SAFEMATH ONCE THE SOLIDITY VERSION IS 0.8.0 OR GREATER

Version 0.8.0 introduces internal overflow checks, so using `SafeMath` is redundant and adds overhead

There are 3 instances of this issue.

[G-25] DUPLICATED REQUIRE()/REVERT() CHECKS SHOULD BE REFACTORED TO A MODIFIER OR FUNCTION

Saves deployment costs

There are 7 instances of this issue.

[G-26] MULTIPLE IF-STATEMENTS WITH MUTUALLY-EXCLUSIVE CONDITIONS SHOULD BE CHANGED TO IF-ELSE STATEMENTS

If two conditions are the same, their blocks should be combined

There is 1 instance of this issue:

```
File: contracts/contracts/core/connext/helpers/ConnextPriceOracle.sol   #1

        if (tokenPrice == 0) {
          tokenPrice = getPriceFromOracle(tokenAddress);
        }
        if (tokenPrice == 0) {
          tokenPrice = getPriceFromDex(tokenAddress);
        }
```

https://github.com/code-423n4/2022-06-connext/blob/4dd6149748b635f95460d4c3924c7e3fb6716967/contracts/contracts/core/connext/helpers/ConnextPrice L92

[G-27] REQUIRE() OR REVERT() STATEMENTS THAT CHECK INPUT ARGUMENTS SHOULD BE AT THE TOP OF THE FUNCTION

Checks that involve constants should come before checks that involve state variables

There are 6 instances of this issue.

[G-28] EMPTY BLOCKS SHOULD BE REMOVED OR EMIT SOMETHING

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be `abstract` and the function signatures be added without any default implementation. If the block is an empty `if`-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (`if(x){}else if(y){...}else{...}` => `if(!x){if(y){...}else{...}}`)

There is 1 instance of this issue:

```
File: contracts/contracts/core/promise/PromiseRouter.sol   #1

132:    receive() external payable {}
```

https://github.com/code-423n4/2022-06-connext/blob/4dd6149748b635f95460d4c3924c7e3fb6716967/contracts/contracts/core/promise/PromiseRouter.sol#L

[G-29] SUPERFLUOUS EVENT FIELDS

`block.timestamp` and `block.number` are added to event information by default so adding them manually wastes gas

There are 4 instances of this issue.

[G-30] USE CUSTOM ERRORS RATHER THAN REVERT()/REQUIRE() STRINGS TO SAVE GAS

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hitby avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas

There are 79 instances of this issue.

[G-31] FUNCTIONS GUARANTEED TO REVERT WHEN CALLED BY NORMAL USERS CAN BE MARKED `PAYABLE`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE`(2),`DUP1`(3),`ISZERO`(3),`PUSH2`(3),`JUMPI`(10),`PUSH1`(3),`DUP1`(3),`REVERT`(0),`JUMPDEST`(1),`POP`(2), which costs an average of about 21 gas per call to the function, in addition to the extra deployment cost

There are 85 instances of this issue.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top