Connext NXTP — Noncustodial Xchain Transfer Protocol

Date	July 2021
Auditors	Martin Ortner, David Oz Kashi, Heiko Fisch
Download	PDF

1 Executive Summary

This report presents the results of our engagement with **Connext** to review their **NXTP protocol for crosschain transfers**.

The review was conducted over two weeks, from **July 26** to **August 6, 2021**, by **Heiko Fisch**, **David Oz Kashi**, and **Martin Ortner**. A total of 25 person-days were spent. The off-chain code components were reviewed on a best effort basis considering the given time constraints.

During the first week, we focused on both the on-chain components of the system and the router (TypeScript) component. During the second week, we continued to focus on the on-chain components and inspected the sdk (TypeScript) component.

From **August 30** to **September 3, 2021**, a revised version of the contract code (`0656436d`), has been reviewed by **Heiko Fisch**, and the report has been updated accordingly. The critical (4.1) and major (4.3) contract issues found during the original audit have been fixed, but a new major issue (4.2) has been introduced.

In **November 2021**, the client has asked us to publicize the report. Meanwhile, the code had undergone some changes since the revised version that was reviewed in August/September (`0656436d`). In particular, the Connext team claims to have fixed issues 4.2 and 4.14. We have not reviewed these fixes or any changes to the codebase after `0656436d`.

1.1 Scope

Our initial review focused on the commit hash `494d07d3707df91658de45e2177a79adc80cf5fe`, and the revised version has commit hash `0656436d654cfe0313fa3c2bbc81aa86232ade16`. The “Resolution” text boxes were added in the August/September review and pertain to the revised version (`0656436d`) unless indicated otherwise (as in 4.2 and the last paragraph of 4.14). The list of files in scope can be found in the Appendix.

2 System Overview

3 Security Specification

This section outlines the system’ s actors and roles and describes some risks, responsibilities, and trust assumptions that are consequences of its design.

3.1 Actors

The relevant actors are listed below with their respective abilities:

- Users (actors that are willing to swap cross-chain assets)
  - deposit the asset to be swapped
  - withdraw the target asset
  - cancel a swap request
- Routers
  - add liquidity to facilitate future swaps
  - remove liquidity
  - deposit the asset to be swapped
  - withdraw the swap source asset
  - cancel a swap request
- Relayers
  - fulfill a swap transaction on behalf of a user
  - cancel a swap transaction on behalf of a user
- Owner
  - add and remove routers
  - add and remove assets

3.2 Risks, responsibilities, and trust assumptions for users, routers, and relayers

- As soon as a user has revealed their `fulfill` signature to anyone (including a relayer or a mempool on the receiving chain), it can be used to `fulfill` on the sending chain, which means the user’ s funds are transferred to the router. The user’ s interest is to `fulfill` on the receiving chain; they might either do that themself or trust that a relayer who will do it for them. Hence, they must reveal their signature *early enough* to give the relayers the possibility to do that before receiving-side expiry. A user might even choose to start so early that they still have time to `fulfill` themself (which might involve buying gas on the receiving chain), in case no relayer picks up the task. Factors like chain congestion have to be taken into account, too.
  The router faces a symmetric risk with regard to chain congestion on the sending chain. The router must therefore choose an expiry on the receiving chain that gives them enough time to fulfill on the sending chain, even if the user’ s signature isn’ t revealed any sooner than expiry on the receiving chain.

- Users must, in their own best interest, never reuse a `transactionId` they have used before — not even across different chains and no matter whether the transaction was successful or not. (See also issue 4.18.)

- A user must never sign a `cancel` while fulfillment on the sending chain is still possible. This is discussed more thoroughly in issue 4.10.

- Users must carefully examine and validate the transaction counterpart on the receiving side, particularly `amount` and `expiry`.
  [The next sentence only applies to the original version of the code, as the shares logic has been removed in the revised version.]
  As outlined in issue 4.3, item 5, the `amount` emitted in `TransactionPrepared` can differ significantly from what a user could expect if the number of shares stored in the `txData` would be converted immediately to a token amount again (which is the relevant information).

- Actions that are supposed to follow a successful on-chain transaction should wait for enough confirmations. For example, the router should only prepare the corresponding receiving-chain transaction when they can be sufficiently sure that the sending-side transaction won’ t vanish in a chain reorganization. Similarly, the user should only reveal the signature when they are confident that the receiving-chain transaction will persist. (See also issue 4.15.)

- Price volatility is a natural risk with commitments to exchange, at a later time, a certain amount of one asset for a certain amount of a different asset; when the transfers actually take place, prices may have deviated considerably. This applies similarly to agreed-upon fees in a volatile asset.

- After the user has sent the sending-chain transaction, the router might not follow through and send the receiving-chain transaction. In that case, the user’ s capital is locked until expiry, and the user had/has to pay the gas costs for `prepare` and `cancel` without getting anything in return. Similarly, if the router does `prepare` on the receiving chain and the user does not reveal their signature, the router’ s funds will be locked until expiry, and the gas for the transaction will be wasted; however, the same is true for the user.

- A transaction sent by a relayer might revert due to unforeseen reasons such as a competing relayer front-running their transaction; in that case, they have to pay the gas costs for the failing transaction but will not get the fee.
  Similarly, since untrusted code may be executed during fulfillment on the receiving chain (e.g., a recipient hook in the token code) gas consumption for the transaction could be higher than anticipated. (Again, this might happen in a front-running manner.) This could incur a loss for the relayer, no matter whether the transaction gets executed and the gas costs are higher than the relayer fee or whether the transaction runs out of gas and reverts.
  [This item has been clarified and extended compared to the original version of the report.]

- Token contracts have to be carefully vetted for compatibility with NXTP. Bugs, privileges of operators, non-standard or weird behavior, and more or less accepted features like blacklisting obviously can have an impact on the protocol and cause lost or stuck funds.

- The `owner` of the `TransactionManager` contract has the privilege to add and remove routers as well as assets. That means routers and assets could be censored at will, even in a front-running or sandwiching manner. Best practices for key management of a privileged account should be followed.

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 TransactionManager - User might steal router’s locked funds

Critical Fixed

Resolution

This issue has been fixed.

Description

`TransactionManager.removeLiquidity` is intended to be restricted for routers only, but in practice, it’s callable by users that had deposited funds to the contract using `TransactionManager.prepare`. A user may initiate a `prepare` transaction, wait for the router to lock his funds (by calling `prepare` on the receiving chain), then the user can call `removeLiquidity`, and `fulfill` (on the receiving chain), thus stealing router’s locked funds while claiming his locked funds back.

Recommendation

Consider using a data structure different than `issuedShares` for storing user deposits. This way, withdrawals by users will only be allowed when calling `TransactionManager.cancel`.

4.2 TransactionManager - Receiver-side check also on sending side

Major Unverified Fix

Resolution

The Connext team claims to have fixed this in commit `4adbfd52703441ee5de655130fc2e0252eae4661`. We have not reviewed this commit or, generally, the codebase at this point.

Description

The functions `prepare`, `cancel`, and `fulfill` in the `TransactionManager` all have a “common part” that is executed on both the sending and the receiving chain and side-specific parts that are only executed either on the sending or on the receiving side.
The following lines occur in `fulfill`’s common part, but this should only be checked on the receiving chain. In fact, on the sending chain, we might even compare amounts of different assets.

code2/packages/contracts/contracts/TransactionManager.sol:L476-L478

```
// Sanity check: fee <= amount. Allow `=` in case of only wanting to execute
// 0-value crosschain tx, so only providing the fee amount
require(relayerFee <= txData.amount, "#F:023");
```

This could prevent a legitimate `fulfill` on the sending chain, causing a loss of funds for the router.

Recommendation

Move these lines to the receiving-side part.

Remark

The `callData` supplied to `fulfill` is not used at all on the sending chain, but the check whether its hash matches `txData.callDataHash` happens in the common part.

code2/packages/contracts/contracts/TransactionManager.sol:L480-L481

```
// Check provided callData matches stored hash
require(keccak256(callData) == txData.callDataHash, "#F:024");
```

In principle, this check could also be moved to the receiving-chain part, allowing the router to save some gas by calling sending-side `fulfill` with empty `callData` and skip the check. Note, however, that the `TransactionFulfilled` event will then also emit the “wrong” `callData` on the sending chain, so the off-chain code has to be able to deal with that if you want to employ this optimization.

4.3 TransactionManager - Flawed shares arithmetic Major Fixed

Resolution

Comment from Connext:

We removed the shares logic completely, and are instead only focusing on standard tokens (i.e. rebasing, inflationary, and deflationary tokens are not supported directly). If users want to transfer non-standard tokens, they do so at their own risk.

Description

To support a wide variety of tokens, the `TransactionManager` uses a per-asset *shares system* to represent fractional ownership of the contract’ s balance in a token. There are several flaws in the shares-related arithmetic, such as:

1. `addLiquidity` and sender-side `prepare` convert asset amounts 1:1 to shares, instead of taking the current value of a share into account. A 1:1 conversion is only appropriate if the number of shares is 0, for example, for the first deposit.
2. The `WadRayMath` library is not used correctly (and maybe not the ideal tool for the task in the first place): `rayMul` and `rayDiv` each operate on two rays (decimal numbers with 27 digits) but are not used according to that specification in `getAmountFromIssuedShares` and `getIssuedSharesFromAmount`. The scaling errors cancel each other out, though.
3. The `WadRayMath` library rounds to the nearest representable number. That might not be desirable for NXTP; for example, converting a token amount to shares and back to tokens might lead to a higher amount than we started with.
4. The `WadRayMath` library reverts on overflows, which might not be acceptable behavior. For instance, a receiver-side `fulfill` might fail due to an overflow in the conversion from shares to a token amount. The corresponding `fulfill` on the sending chain might very well succeed, though, and it is possible that, at a later point, the receiver-side transaction can be canceled. (Note that canceling does not involve actually converting shares into a token amount, but the calculation is done anyway for the event.)
5. The `amount` emitted in the `TransactionPrepared` event on the receiving chain can, depending on the granularity of the shares, differ considerably from what a user can expect to receive when the shares are converted back into tokens. The reason for this is the double conversion from the initial token amount — which is emitted — to shares and, later, back to tokens.
6. Special cases might have to be taken into account. As a more subtle example, converting a non-zero token amount to shares is not possible (or at least not with the usual semantics) if the contract’ s balance is zero, but the number of already existing shares is strictly greater than zero, as any number of shares will give you back less than the original amount. Whether this situation is possible depends on the token contract.

Recommendation

The shares logic was added late to the contract and is still in a pretty rough shape. While providing a full-fledged solution is beyond the scope of this review, we hope that the points raised above provide pointers and guidelines to inform a major overhaul.

4.4 Router - handleMetaTxRequest - gas griefing / race conditions / missing validations / free meta transactions Major

Description

There’ s a comment in `handleMetaTxRequest` that asks whether data needs to be validated before interacting with the contract and the answer is yes, always, or else this opens up a gas griefing vector on the router side.

For example, someone might flood broadcast masses of metaTx requests (`*.*.metatx`) and all online routers will race to call `TransactionManager.fulfill()`. Even if only one transaction should be able to successfully go through all the others will loose on gas (until up to the first require failing).

Given that there is no rate limiting and it is a broadcast that is very cheap to perform on the client-side (I can just spawn a lot of nodes spamming messages) this can be very severe, keeping the router busy sending transactions that are deemed to fail until the routers balance falls below the min gas limit configured.

Even if the router would check the contracts current state first (performing read-only calls that can be done offline) to check if the transaction has a chance to succeed, it will still compete in a race for the current block (mempool).

Examples

code/packages/router/src/handler.ts:L459-L477

```
const fulfillData: MetaTxFulfillPayload = data.data;
// Validate that metatx request matches with known data about fulfill
// Is this needed? Can we just submit to chain without validating?
// Technically this is ok, but perhaps we want to validate only for our own
// logging purposes.
// Would also be bad if router had no gas here
// Next, prepare the tx object
// - Get chainId from data
// - Get fulfill fee from data and validate it covers gas
// - etc.
// Send to txService
// Update metrics

// TODO: make sure fee is something we want to accept

this.logger.info({ method, methodId, requestContext, chainId, data }, "Submitting tx");
const res = await this.txManager
  .fulfill(
    chainId,
```

Recommendation

For state-changing transactions that actually cost gas there is no way around implementing strict validation whenever possible and avoid performing the transaction in case validation fails. Contract state should always be validated before issuing new online transactions but this might not fix the problem that routers still compete for their transaction to be included in the next block (mempool not monitored). The question therefore is, whether it would be better to change the metaTX flow to have a router confirm that they will send the tx via the messaging service first so others know they do not even have to try to send it. However, even this scenario may allow to DoS the system by maliciously responding with such a method.

In general, there’ re a lot of ways to craft a message that forces the router to issue an on-chain transaction that may fail with no consequences for the sender of the metaTx message.

Additionally, the `relayerFee` is currently unchecked which may lead to the router loosing funds because they effectively accept zero-fee relays.

As noted in issue 4.6, the missing return after detecting that the metatx is destined for a TransactionManager that is not supported allows for explicit gas griefing attacks (deploy a fake TransactionManager.fulfill that mines all the gas for a beneficiary).

The `contract` methods should additionally validate that the sender balance can cover for the gas required to send the transaction.

4.5 Router - subgraphLoop may process transactions the router was not configured for (code fragility) Major

Description

`subgraphLoop` gets all sending transactions for the router, chain, status triplet.

code/packages/router/src/subgraph.ts:L155-L159

```
allSenderPrepared = await sdk.GetSenderTransactions({
  routerId: this.routerAddress.toLowerCase(),
  sendingChainId: chainId,
  status: TransactionStatus.Prepared,
});
```

and then sorts the results by receiving chain id. **Note** that this keeps track of chainID’ s the router was not configured for.

code/packages/router/src/subgraph.ts:L168-L176

```
// create list of txIds for each receiving chain
const receivingChains: Record<string, string[]> = {};
allSenderPrepared.router?.transactions.forEach(({ transactionId, receivingChainId }) => {
  if (receivingChains[receivingChainId]) {
    receivingChains[receivingChainId].push(transactionId);
  } else {
    receivingChains[receivingChainId] = [transactionId];
  }
});
```

In a next step, transactions are resolved from the various chains. This filters out chainID’ s the router was not configured for (and just returns an empty array), however, the `GetTransactions` query assumes that `transactionID`’ s are unique across the subgraph which might not be true!

code/packages/router/src/subgraph.ts:L179-L193

```
let correspondingReceiverTxs: any[];
try {
  const queries = await Promise.all(
    Object.entries(receivingChains).map(async ([cId, txIds]) => {
      const _sdk = this.sdks[Number(cId)];
      if (!_sdk) {
        this.logger.error({ chainId: cId, method, methodId }, "No config for chain, this should not ha
        return [];
      }
      const query = await _sdk.GetTransactions({ transactionIds: txIds.map((t) => t.toLowerCase()) });
      return query.transactions;
    }),
  );
  correspondingReceiverTxs = queries.flat();
} catch (err) {
```

In the last step, all chainID’ s (even the one’ s the router was not configured for) are iterated again (which might be unnecessary). TransactionID’ s are loosely matched from the previously flattened results from all the various chains. Since transactionID’ s don’ t necessarily need to be unique across chains or within the chain, it is likely that the subsequent matching of transactionID’ s (`correspondingReceiverTxs.find`) returns more than 1 entry. However, `find()` just returns the first item and covers up the fact that there might be multiple matches. Also, since the code returned an empty array for chains it was not configured for, the find will return `undef` satisfying the `!corresponding` branch and fire an `SenderTransactionPrepared` triggering the handler to perform an on-chain action that will most definitely fail at some point.

Recommendation

The code in this module is generally very fragile. It is based on assumptions that can likely be exploited by a third party re-using transactionID’ s (or other values). It is highly recommended to rework the code making it more resilient to potential corner cases.

- Filter receivingChains for chainID’ s that are not supported by the router
- Avoid having to integrating the `allSenderPrepared` array twice and use a filtered list instead
- Change the very broad query in `_sdk.GetTransactions()` that assumes transactionID’ s are unique across all chains to a specific query that selects transactions specific to the chain and this router. The more specific the better!
- When matching the transactions also match the source/receiver chains instead of only matching the transactionID. Additionally, check if more than one entry matches the condition instead of silently taking the first result (this is what `array.find()` does)

Also see issue 5.2

4.6 Router - handler reports an error condition but continues execution instead of aborting it Major

Description

There are some code paths that detect and log an error but then continue the execution flow instead of returning the error condition to the caller. This may allow for a variety of griefing vectors (e.g. gas griefing).

Examples

- reports an error because the received address does not match our configured transaction manager, but then proceeds. This means the router would accept a transaction manager it was not configured for.

code/packages/router/src/handler.ts:L448-L458

```
if (utils.getAddress(data.to) !== utils.getAddress(chainConfig.transactionManagerAddress)) {
  const err = new HandlerError(HandlerError.reasons.ConfigError, {
    requestContext,
    calling: "chainConfig.transactionManagerAddress",
    methodId,
    method,
    configError: `Provided transactionManagerAddress does not map to our configured transactionManager
  });
  this.logger.error({ method, methodId, requestContext, err: err.toJson() }, "Error in config");
}
```

- If `chainConfig` is `undef` this should return or else it will bail later

code/packages/router/src/handler.ts:L436-L445

```
if (!chainConfig) {
  const err = new HandlerError(HandlerError.reasons.ConfigError, {
    requestContext,
    calling: "getConfig",
    methodId,
    method,
    configError: `No chainConfig for ${chainId}`,
  });
  this.logger.error({ method, methodId, requestContext, err: err.toJson() }, "Error in config");
}
```

- if data is not `fulfill` this silently returns, while it should probably raise an error instead (unexpected message)

code/packages/router/src/handler.ts:L447-L447

```
if (data.type === "Fulfill") {
```

Recommendation

- Implement strict validation of untrusted data.
- Be explicit and raise error conditions on unexpected messages (e.g. `type` is not `fulfill`) instead of silently skipping the message.
- Add the missing returns after reporting an error instead of continuing the execution flow on errors.

4.7 Router - spawns unauthenticated admin API endpoint listening

on all interfaces Major

Description

- unauthenticated
- listening on allips

pot. allows any local or remote unpriv user with access to the endpoint to steal the routers liquidity `/remove-liquidity -> req.body.recipientAddress`

Examples

code/packages/router/src/index.ts:L123-L130

```
server.listen(8080, "0.0.0.0", (err, address) => {
  if (err) {
    console.error(err);
    process.exit(1);
  }
  console.log(`Server listening at ${address}`);
});
```

Recommendation

- require authentication
- should only bind to localhost by default

4.8 TODO comments should be resolved Medium

Description

As part of the process of bringing the application to production readiness, dev comments (especially TODOs) should be resolved. In many cases, these comments indicate a missing functionality that should be implemented, or some missing necessary validation checks.

4.9 TransactionManager - Missing `nonReentrant` modifier on `removeLiquidity` Medium Fixed

Resolution
This issue has been fixed.

Description

The `removeLiquidity` function does not have a `nonReentrant` modifier.

code/packages/contracts/contracts/TransactionManager.sol:L274-L329

```solidity
/**
 * @notice This is used by any router to decrease their available
 *         liquidity for a given asset.
 * @param shares The amount of liquidity to remove for the router in shares
 * @param assetId The address (or `address(0)` if native asset) of the
 *                asset you're removing liquidity for
 * @param recipient The address that will receive the liquidity being removed
 */
function removeLiquidity(
  uint256 shares,
  address assetId,
  address payable recipient
) external override {
  // Sanity check: recipient is sensible
  require(recipient != address(0), "#RL:007");

  // Sanity check: nonzero shares
  require(shares > 0, "#RL:035");

  // Get stored router shares
  uint256 routerShares = issuedShares[msg.sender][assetId];

  // Get stored outstanding shares
  uint256 outstanding = outstandingShares[assetId];

  // Sanity check: owns enough shares
  require(routerShares >= shares, "#RL:018");

  // Convert shares to amount
  uint256 amount = getAmountFromIssuedShares(
    shares,
    outstanding,
    Asset.getOwnBalance(assetId)
  );

  // Update router issued shares
  // NOTE: unchecked due to require above
  unchecked {
    issuedShares[msg.sender][assetId] = routerShares - shares;
  }

  // Update the total shares for asset
  outstandingShares[assetId] = outstanding - shares;

  // Transfer from contract to specified recipient
  Asset.transferAsset(assetId, recipient, amount);

  // Emit event
  emit LiquidityRemoved(
    msg.sender,
    assetId,
    shares,
    amount,
    recipient
  );
}
```

Assuming we’re dealing with a token contract that allows execution of third-party-supplied code, that means it is possible to leave the `TransactionManager` contract in one of the functions that call into the token contract and then reenter via `removeLiquidity`.

Alternatively, we can leave the contract in `removeLiquidity` and reenter through an arbitrary external function, even if it has a `nonReentrant` modifier.

Example

Assume a token contract allows the execution of third-party-supplied code in its `transfer` function *before* the actual balance change takes place. If a router calls `removeLiquidity` with half of their shares and then, in a reentering `removeLiquidity` call, supplies the other half of their shares, they will receive more tokens than if they had liquidated all their shares at once because the reentering call occurs after the (first half of the) shares have been burnt but before the corresponding amount of tokens has actually been transferred out of the contract, leading to an artificially increased share value in the reentering call.
Similarly, reentering the contract with a `fulfill` call on the receiving chain instead of a second `removeLiquidity` would transfer too many tokens to the recipient due to the artificially inflated share value.

Recommendation

While tokens that behave as described in the example might be rare or not exist at all, caution is advised when integrating with unknown tokens or calling untrusted code in general. We strongly recommend adding a `nonReentrant` modifier to `removeLiquidity`.

4.10 TransactionManager - Relayer may use user’s cancel after expiry signature to steal user’s funds by colluding with a router Medium Acknowledged

Resolution

This has been acknowledged by the Connext team. As discussed below in the “Recommendation”, it is not a flaw in the contracts *per se* but rather a high-risk situation caused by cancellation signatures working on both sender and receiver side. As immediate mitigation, sender-side cancellation via signature has been removed completely. The “signature rules” explained below still apply and have to be followed.

Description

Users that are willing to have a lower trust dependency on a relayer should have the ability to opt-in **only** for the service that allows the relayer to withdraw back users’ funds from the sending chain after expiry. However, in practice, a user is forced to opt-in for the service that refunds the router before the expiry, since the same signature is used for both services (lines 795,817 use the same signature).

Let’s consider the case of a user willing to call `fulfill` on his own, but to use the relayer only to withdraw back his funds from the sending chain after expiry. In this case, the relayer can collude with the router and use the user’s `cancel` signature (meant for withdrawing **his** only after expiry) as a front-running transaction for a user call to `fulfill`. This way the router will be able to withdraw both his funds and the user’s funds since the user’s `fulfill` signature is now public data residing in the mem-pool.

Examples

code/packages/contracts/contracts/TransactionManager.sol:L795-L817

```
        require(msg.sender == txData.user || recoverSignature(txData.transactionId, relayerFee, "cancel"

        Asset.transferAsset(txData.sendingAssetId, payable(msg.sender), relayerFee);
      }

      // Get the amount to refund the user
      uint256 toRefund;
      unchecked {
        toRefund = amount - relayerFee;
      }

      // Return locked funds to sending chain fallback
      if (toRefund > 0) {
        Asset.transferAsset(txData.sendingAssetId, payable(txData.sendingChainFallback), toRefund);
      }
    }

  } else {
    // Receiver side, router liquidity is returned
    if (txData.expiry >= block.timestamp) {
      // Timeout has not expired and tx may only be cancelled by user
      // Validate signature
      require(msg.sender == txData.user || recoverSignature(txData.transactionId, relayerFee, "cancel",
```

Recommendation

The crucial point here is that the user must never sign a “cancel” that could be used on the receiving chain while fulfillment on the sending chain is still a possibility.
Or, to put it differently: A user may only sign a “cancel” that is valid on the receiving chain after sending-chain expiry or if they never have and won’t ever sign a “fulfill” (or at least won’t sign until sending-chain expiry — but it is pointless to sign a “fulfill” after that, so “never” is a reasonable simplification).
Or, finally, a more symmetric perspective on this requirement: If a user has signed “fulfill”, they must not sign a receiving-chain-valid “cancel” until sending-chain expiry, and if they have signed a receiving-chain-valid “cancel”, they must not sign a “fulfill” (until sending-chain expiry).

In this sense, “cancel” signatures that are valid on the receiving chain are dangerous, while sending-side cancellations are not. So the principle stated in the previous paragraph might be easier to follow with different signatures for sending- and receiving-chain cancellations.

4.11 Router - handleSenderPrepare - missing validation, unchecked bidExpiry, unchecked expiry, unchecked chainids/swaps, race conidtions Medium

Description

This finding highlights a collection of issues with the `handleSenderPrepare` method. The code and coding style appears fragile. Validation should be strictly enforced and protective measures against potential race conditions should be implemented.

The following list highlights individual findings that contribute risk and therefore broaden the attack surface of this method:

- unchecked `bidExpiry` might allow using bids even after expiration.

code/packages/router/src/handler.ts:L612-L626

```
.andThen(() => {
  // TODO: anything else? seems unnecessary to validate everything
  if (!BigNumber.from(bid.amount).eq(amount) || bid.transactionId !== txData.transactionId) {
    return err(
      new HandlerError(HandlerError.reasons.PrepareValidationError, {
        method,
        methodId,
        calling: "",
        requestContext,
        prepareError: "Bid params not equal to tx data",
      }),
    );
  }
  return ok(undefined);
});
```

- unchecked `txdata.expiry` might lead to router preparing for an already expired prepare. However, this is rather unlikely easily exploitable as the data source is a subgraph.
- a bid might not be fulfillable anymore due to changes to the router (e.g. removing a chainconfig or assets) but the router would still attempt it. Make sure to always verify chainid/assets/the configured system parameters.
- potential race condition. make sure to lock the txID in the beginning.

code/packages/router/src/handler.ts:L663-L669

```
// encode the data for contract call
// Send to txService
this.receiverPreparing.set(txData.transactionId, true);
this.logger.info(
  { method, methodId, requestContext, transactionId: txData.transactionId },
  "Sending receiver prepare tx",
);
```

- Note that transactionID’s as they are used in the system must be unique across chains.

4.12 Router - handleNewAuction - fragile code Medium

Description

This finding highlights a collection of issues with the `handleNewAuction`. The code and coding style appears fragile. Validation should be strictly enforced, debugging code should be removed or disabled in production and protective measures should be taken from abusive clients.

The following list highlights individual findings that contribute risk and therefore broaden the attack surface of this method:

- router bids on zero-amount requests (this will fail later when calling the contract, thus a potential gas griefing attack vector)

code/packages/router/src/handler.ts:L197-L201

```
// validate that assets/chains are supported and there is enough liquidity
// and gas on both sender and receiver side.
// TODO: will need to track this offchain
const amountReceived = mutateAmount(amount);
```

- duplicate constant var assignment (subfunction const shadowing and unchecked initial config!)

code/packages/router/src/handler.ts:L202-L204

```
const config = getConfig();
const sendingConfig = config.chainConfig[sendingChainId];
const receivingConfig = config.chainConfig[receivingChainId];
```

code/packages/router/src/handler.ts:L231-L240

```
// validate config
const config = getConfig();
const sendingConfig = config.chainConfig[sendingChainId];
const receivingConfig = config.chainConfig[receivingChainId];
if (
  !sendingConfig.providers ||
  sendingConfig.providers.length === 0 ||
  !receivingConfig.providers ||
  receivingConfig.providers.length === 0
) {
```

- actual estimated gas required to fuel transaction is never checked. current balance might be outdated, especially in race condition scenarios.

code/packages/router/src/handler.ts:L315-L318

```
.andThen((balances) => {
  const [senderBalance, receiverBalance] = balances as BigNumber[];
  if (senderBalance.lt(sendingConfig.minGas) || receiverBalance.lt(receivingConfig.minGas)) {
    return errAsync(
```

- remove debug code from production build (`dry-run`)

code/packages/router/src/handler.ts:L194-L194

```
dryRun,
```

code/packages/router/src/handler.ts:L385-L385

```
this.messagingService.publishAuctionResponse(inbox, { bid, bidSignature: dryRun ? undefined : bidSigna
```

- signer address might be different for different chains

code/packages/router/src/handler.ts:L290-L312

```
return combine([
  ResultAsync.fromPromise(
    this.txService.getBalance(sendingChainId, this.signer.address),
    (err) =>
      new HandlerError(HandlerError.reasons.TxServiceError, {
        calling: "txService.getBalance => sending",
        method,
        methodId,
        requestContext,
        txServiceError: jsonifyError(err as NxtpError),
      }),
  ),
  ResultAsync.fromPromise(
    this.txService.getBalance(receivingChainId, this.signer.address),
    (err) =>
      new HandlerError(HandlerError.reasons.TxServiceError, {
        calling: "txService.getBalance => receiving",
        method,
        methodId,
        requestContext,
        txServiceError: jsonifyError(err as NxtpError),
      }),
  ),
```

- no rate limiting. potential DoS vector when someone floods the node with auction requests (significant work to be done, handler is async, will trigger a reply message). user might force the router to sign the same message multiple times.

- missing validation of bid parameters (expiriy within valid range, …)

4.13 Router - Cancel is not implemented Medium

Description

Canceling of failed/expired swaps does not seem to be implemented in the router. This may allow a user to trick the router into preparing all its funds which will not automatically be reclaimed after expiration (router DoS).

Examples

- cancelExpired is never called

code/packages/sdk/src/sdk.ts:L873-L885

```
// TODO: this just cancels a transaction, it is misnamed, has nothing to do with expiries
public async cancelExpired(cancelParams: CancelParams, chainId: number): Promise<providers.Transactior
  const method = this.cancelExpired.name;
  const methodId = getRandomBytes32();
  this.logger.info({ method, methodId, cancelParams, chainId }, "Method started");
  const cancelRes = await this.transactionManager.cancel(chainId, cancelParams);
  if (cancelRes.isOk()) {
    this.logger.info({ method, methodId }, "Method complete");
    return cancelRes.value;
  } else {
    throw cancelRes.error;
  }
}
```

- disabled code

code/packages/router/src/handler.ts:L719-L733

```
  "Do not cancel ATM, figure out why we are in this case first",
);
// const cancelRes = await this.txManager.cancel(txData.sendingChainId, {
//   txData,
//   signature: "0x",
//   relayerFee: "0",
// });
// if (cancelRes.isOk()) {
//   this.logger.warn(
//     { method, methodId, transactionHash: cancelRes.value.transactionHash },
//     "Cancelled transaction",
//   );
// } else {
//   this.logger.error({ method, methodId }, "Could not cancel transaction after error ");
// }
```

Recommendation

Implement the cancel flow.

4.14 TransactionManager.prepare - Possible griefing/denial of service by front-running Medium Unverified Fix

Resolution

Comment from Connext:

> We see this as a highly unlikely attack vector and have chosen not to mitigate it, but it is possible. Users can always and easily generate a new key prepare from a new account, and performing this attack will always cost gas and some dust amount. Further, adding in the suggested `require(msg.sender == invariantData.user)` will lock out many contract-based use cases and requiring an additional signature/user interaction (`auth`, `approve`, `prepare`, `fulfill`) is not desirable.

Indeed, since the `user` has to sign messages, it has to be an EOA, and, consequently, the suggested solution would exclude contracts from calling `prepare`. A slight modification of the recommendation should work, though: Instead of checking `msg.sender == invariantData.user`, add a new member `initiator` (or `msgSender` or something similar) to the `InvariantTransactionData` struct, and check `msg.sender == invariantData.initiator` in the `prepare` function. That would fix the issue and still allow `prepare` calls from a contract.

The Connext team claims to have implemented this solution in commit `6811bb2681f44f34ce28906cb842db49fb73d797`. We have not reviewed this commit or, generally, the codebase at this point.

Description

A call to `TransactionManager.prepare` might be front-run with a transaction using the same `invariantData` but with a different `amount` and/or `expiry` values. By choosing a tiny amount of assets, the attacker may prevent the user from locking his original desired amount. The attacker can repeat this process for any new `transactionId` presented by the user, thus effectively denying the service for him.

Recommendation

Consider adding a `require(msg.sender == invariantData.user)` restriction to `TransactionManager.prepare`.

4.15 Router - Provide and enforce safe defaults (config) Medium

Description

Chain `confirmations` default to `1` which is not safe. In case of a re-org the router might (temporarily) get out of sync with the chain and perform actions it should not perform. This may put funds at risk.

Examples

the schema requires an unsafe minimum of 1 confirmation

code/packages/router/src/config.ts:L33-L36

```
export const TChainConfig = Type.Object({
  providers: Type.Array(Type.String()),
  confirmations: Type.Number({ minimum: 1 }),
  subgraph: Type.String(),
```

the default configuration uses 1 confirmation

code/packages/router/config.json.example:L1-L17

```
{
  "adminToken": "blahblah",
  "chainConfig": {
    "4": {
      "providers": ["https://rinkeby.infura.io/v3/"],
      "confirmations": 1,
      "subgraph": "https://api.thegraph.com/subgraphs/name/connext/nxtp-rinkeby"
    },
    "5": {
      "providers": ["https://goerli.infura.io/v3/"],
      "confirmations": 1,
      "subgraph": "https://api.thegraph.com/subgraphs/name/connext/nxtp-goerli"
    }
  },
  "logLevel": "info",
  "mnemonic": "candy maple cake sugar pudding cream honey rich smooth crumble sweet treat"
}
```

Recommendation

Give guidance, provide and enforce safe defaults.

4.16 ProposedOwnable - two-step ownership transfer should be confirmed by the new owner Medium Fixed

Resolution

All recommendations given below have been implemented. In addition to that, the privilege to manage assets and the privilege to manage routers can now be renounced separately.

Description

In order to avoid losing control of the contract, the two-step ownership transfer should be confirmed by the new owner's address instead of the current owner.

Examples

- `acceptProposedOwner` is restricted to `onlyOwner` while ownership should be accepted by the newOwner

code/packages/contracts/contracts/ProposedOwnable.sol:L89-L96

```
/**
  * @notice Transfers ownership of the contract to a new account (`newOwner`).
  * Can only be called by the current owner.
  */
function acceptProposedOwner() public virtual onlyOwner {
  require((block.timestamp - _proposedTimestamp) > _delay, "#APO:030");
  _setOwner(_proposed);
}
```

- move `renounced()` to `ProposedOwnable` as this is where it logically belongs to

code/packages/contracts/contracts/TransactionManager.sol:L160-L162

```
function renounced() public view override returns (bool) {
  return owner() == address(0);
}
```

- `onlyOwner` can directly access state-var `_owner` instead of spending more gas on calling `owner()`

code/packages/contracts/contracts/ProposedOwnable.sol:L76-L79

```
modifier onlyOwner() {
    require(owner() == msg.sender, "#OO:029");
    _;
}
```

Recommendation

- `onlyOwner` can directly access `_owner` (gas optimization)
- add a method to explicitly renounce ownership of the contract
- move `TransactionManager.renounced()` to `ProposedOwnable` as this is where it logically belongs to
- change the access control for `acceptProposedOwner` from `onlyOwner` to `require(msg.sender == _proposed)` (new owner).

4.17 FulfillInterpreter - Wrong order of actions in fallback handling Minor

Description

When a transaction with a `callTo` that is not `address(0)` is fulfilled, the funds to be withdrawn on the user's behalf are first transferred to the `FulfillInterpreter` instance that is associated with this `TransactionManager` instance. After that, `execute` is called on that interpreter instance, which, in turn, tries to make a call to `callTo`. If that call reverts or isn't made in the first place because `callTo` is not a contract address, the funds are transferred directly to the `receivingAddress` in the transaction (which becomes `fallbackAddress` in `execute`); otherwise, it's the called contract's task to transfer the previously approved funds from the interpreter.

code2/packages/contracts/contracts/interpreters/FulfillInterpreter.sol:L68-L90

```
bool isNative = LibAsset.isNativeAsset(assetId);
if (!isNative) {
  LibAsset.increaseERC20Allowance(assetId, callTo, amount);
}

// Check if the callTo is a contract
bool success;
bytes memory returnData;
if (Address.isContract(callTo)) {
  // Try to execute the callData
  // the low level call will return `false` if its execution reverts
  (success, returnData) = callTo.call{value: isNative ? amount : 0}(callData);
}

// Handle failure cases
if (!success) {
  // If it fails, transfer to fallback
  LibAsset.transferAsset(assetId, fallbackAddress, amount);
  // Decrease allowance
  if (!isNative) {
    LibAsset.decreaseERC20Allowance(assetId, callTo, amount);
  }
}
```

For the fallback scenario, i.e., the call isn't executed or fails, the funds are first transferred to `fallbackAddress`, and the previously increased allowance is decreased after that. If the token supports it, the recipient of the direct transfer could try to exploit that the approval hasn't been revoked yet, so the logically correct order is to decrease the allowance first and transfer the funds later. However, it should be noted that the `FulfillInterpreter` should, at any point in time, only hold the funds that are supposed to be transferred as part of the current transaction; if there are any excess funds, these are leftovers from a previous failure to withdraw everything that could have been withdrawn, so these can be considered up for grabs. Hence, this is only a minor issue.

Recommendation

We recommend reversing the order of actions for the fallback case: Decrease the allowance first, and transfer later. Moreover, it would be better to increase the allowance only in case a call will actually be made, i.e., if `Address.isContract(callTo)` is `true`.

Remark

This issue was already present in the original version of the code but was missed initially and only found during the re-audit.

4.18 FulfillInterpreter - `Executed` event can't be linked to `TransactionFulfilled` event Minor Fixed

Resolution

This issue has been fixed. We'd like to point out, though:

1. Based on the data emitted by the `TransactionFulfilled` event, it is currently not possible to distinguish between:
   (A) No call to `callTo` has been made because the address didn’t contain code.
   (B) Address `callTo` did contain code, a call was made, and it failed with empty return data.
   If this distinction seems relevant, an additional `bool` should be returned from `FulfillInterpreter.execute` and emitted in `TransactionFulfilled`, indicating which of the two scenarios were encountered.
2. The `Executed` event isn’t needed anymore and could be removed.

Description

When a transaction with a `callTo` that is not `address(0)` is fulfilled, the funds to be withdrawn on the user’s behalf are first transferred to the `FulfillInterpreter` instance that is associated with this `TransactionManager` instance. After that, `execute` is called on that interpreter instance, which, in turn, tries to make a call to `callTo`. If that call reverts, the funds are transferred directly to the `receivingAddress` in the transaction; otherwise, it’s the called contract’s task to transfer the (previously approved) funds from the interpreter. In any case, at the end of `execute` an `Executed` event is emitted that, along with some other transaction data passed in to `execute`, emits whether the call to `callTo` was successful (i.e., didn’t revert) and the return data from that call. The `transactionId` is emitted too; however, that is not necessarily sufficient to link the `Executed` event unambiguously to the `TransactionFulfilled` event emitted in the `fulfill` call — or to the full transaction data, for that matter.

While it is in the user’s best interest not to reuse a `transactionId` they have used before, unique transaction IDs are not *enforced*, and a user seeking to wreak havoc might choose to reuse an ID if it helps them accomplish their goal. In this case, event-monitoring software might get confused by several `Executed` events with the same `transactionId` and not be able to match the event with its `TransactionFulfilled` counterpart.

Recommendation

Rather than emitting a separate `Executed` event — which needs to be linked to its corresponding `TransactionFulfilled` event — it might be better to return `success` and `returnData` to the caller and then emit this information also in the `TransactionFulfilled` event. If you also include `toSend`, everything that is currently emitted across two different events will be part of the `TransactionFulfilled` data; however, `toSend` is not really necessary, as it can be inferred from `amount` and `relayerFee`, which are both emitted anyway.

Generally, the following rules apply to transaction IDs:

1. A user must, in their own best interest, never reuse a `transactionId` they have used before — not even across different chains and no matter whether the transaction was successful or not.
2. This per-user uniqueness of transaction IDs is not enforced, though — not even per `TransactionManager` deployment. Hence, the code may not rely on this assumption, and no harm must come from a reused transaction ID for the system or anyone else than the user who

reused the ID.

4.19 Sdk.finishTransfer - missing validation Minor

Description

`Sdk.finishTransfer` should validate that the router that locks liquidity in the receiving chain, should be the same router the user had committed to in the sending chain.

4.20 FulfillInterpreter - Missing check whether `callTo` address contains code Minor Fixed

Resolution

This issue has been fixed.

Description

The receiver-side `prepare` checks whether the `callTo` address is either zero or a contract:

code/packages/contracts/contracts/TransactionManager.sol:L466-L470

```
// Check that the callTo is a contract
// NOTE: This cannot happen on the sending chain (different chain
// contexts), so a user could mistakenly create a transfer that must be
// cancelled if this is incorrect
require(invariantData.callTo == address(0) || Address.isContract(invariantData.callTo), "#P:031");
```

However, as a contract may `selfdestruct` and the check is not repeated later, there is no guarantee that `callTo` still contains code when the call to this address (assuming it is non-zero) is actually executed in `FulfillInterpreter.execute`:

code/packages/contracts/contracts/interpreters/FulfillInterpreter.sol:L71-L82

```
// Try to execute the callData
// the low level call will return `false` if its execution reverts
(bool success, bytes memory returnData) = callTo.call{value: isEther ? amount : 0}(callData);

if (!success) {
  // If it fails, transfer to fallback
  Asset.transferAsset(assetId, fallbackAddress, amount);
  // Decrease allowance
  if (!isEther) {
    Asset.decreaseERC20Allowance(assetId, callTo, amount);
  }
}
```

As a result, if the contract at `callTo` self-destructs between `prepare` and `fulfill` (both on the receiving chain), `success` will be `true`, and the funds will probably be lost to the user.

A user could currently try to avoid this by checking that the contract still exists before calling `fulfill` on the receiving chain, but even then, they might get front-run by `selfdestruct`, and the situation is even worse with a relayer, so this provides no reliable protection.

Recommendation

Repeat the `Address.isContract` check on `callTo` before making the external call in `FulfillInterpreter.execute` and send the funds to the `fallbackAddress` if the result is `false`.

It is, perhaps, debatable whether the check in `prepare` should be kept or removed. In principle, if the contract gets deployed between `prepare` and `fulfill`, that is still soon enough. However, if the `callTo` address doesn't have code at the time of `prepare`, this seems more likely to be a mistake than a "late deployment". So unless there is a demonstrated use case for "late deployments", failing in `prepare` (even though it's receiver-side) might still be the better choice.

Remark

It should be noted that an unsuccessful call, i.e., a revert, is the only behavior that is recognized by `FulfillInterpreter.execute` as failure. While it is prevalent to indicate failure by reverting, this doesn't *have to* be the case; a well-known example is an ERC20 token that indicates a failing transfer by returning `false`.
A user who wants to utilize this feature has to make sure that the called contract behaves accordingly; if that is not the case, an intermediary contract may be employed, which, for example, reverts for return value `false`.

4.21 TransactionManager - Adherence to EIP-712 Minor Won't Fix

Resolution

Comment from Connext:

> We did not fully adopt EIP712 because hardware wallet support is *still* not universal. Additionally, we chose not to address this issue in the recommended fashion (using `address(this), block.chainId`) because the `fulfill` signature must be usable across both the sending and receiving chain. Instead, we made sure the `transactionManagerReceivingAddress, receivingChainId` was signed.

We advise users of the system not to use their key and address for other systems that operate with signed messages unless they can rule out the possibility of replay attacks. Regarding the signed `receivingChainId` and `receivingChainTxManagerAddress`, we'd like to mention that even for receiver-side fulfillment, these are not verified against the current chain ID and address of the contract.

Description

`fulfill` function requires the user signature on a `transactionId`. While currently, the user SDK code is using a cryptographically secured pseudo-random function to generate the `transactionId`

, it should not be counted upon and measures should be placed on the smart-contract level to ensure replay-attack protection.

Examples

code/packages/contracts/contracts/TransactionManager.sol:L918-L933

```
function recoverSignature(
  bytes32 transactionId,
  uint256 relayerFee,
  string memory functionIdentifier,
  bytes calldata signature
) internal pure returns (address) {
  // Create the signed payload
  SignedData memory payload = SignedData({
    transactionId: transactionId,
    relayerFee: relayerFee,
    functionIdentifier: functionIdentifier
  });

  // Recover
  return ECDSA.recover(ECDSA.toEthSignedMessageHash(keccak256(abi.encode(payload))), signature);
}
```

Recommendation

Consider adhering to EIP-712, or at least including `address(this), block.chainId` as part of the data signed by the user.

4.22 TransactionManager - Hard-coded chain ID might lead to problems after a chain split Minor Pending

Resolution

The recommendation below has been implemented, but the current codebase doesn’ t handle chain splits correctly. On the chain that gets a new chain ID, funds may be lost or frozen.

More specifically, after a chain split, we may find ourselves in the situation that the current chain ID is neither the `sendingChainId` nor the `receivingChainId` stored in the invariant transaction data. If that is the case, we’ re on the chain that got a new chain ID. `fulfill` should always revert in this situation, but cancellation should be possible to release locked funds. We don’ t know, however, whether we should send the funds back to the user (that is, we’ re on a fork of the sending chain) or whether they should be given back to the router (that is, we’ re on a fork of the receiving chain). Our recommendation to solve this is to store in the variant transaction data explicitly whether this is the sending chain or the receiving chain; with this information, we can disambiguate the situation and implement `cancel` correctly.

Description

The ID of the chain on which the contract is deployed is supplied as a constructor argument and stored as an `immutable` state variable:

code/packages/contracts/contracts/TransactionManager.sol:L104-L107

```
/**
 * @dev The chain id of the contract, is passed in to avoid any evm issues
 */
uint256 public immutable chainId;
```

code/packages/contracts/contracts/TransactionManager.sol:L125-L128

```
constructor(uint256 _chainId) {
  chainId = _chainId;
  interpreter = new FulfillInterpreter(address(this));
}
```

Hence, `chainId` can never change, and even after a chain split, both contracts would continue to use the same chain ID. That can have undesirable consequences. For example, a transaction that was prepared before the split could be fulfilled on both chains.

Recommendation

It would be better to query the chain ID directly from the chain via `block.chainId`. However, the development team informed us that they had encountered problems with this approach as some chains apparently are not implementing this correctly. They resorted to the method described above, a constructor-supplied, hard-coded value. For chains that do indeed not inform correctly about their chain ID, this is a reasonable solution. However, for the reasons outlined above, we still recommend querying the chain ID via `block.chainId` *for chains that do support that* — which should be the vast majority — and using the fallback mechanism only when necessary.

4.23 Router - handling of native assetID (`0x000..00`, e.g. `ETH`) not implemented Minor

Description

`Contract.ts` does not implement the native Asset (`0x000...000`; `ETH`). Transaction value is hardcoded to zero. `approveTokensIfNeeded` will likely fail as it will attempt to contract call `0x0` and there is inconsistent use of default values (`0` vs BigInt `const.Zero`).

Additionally, `handleSenderPrepare` does not manage approvals for `ERC20` transfers.

Examples

- harcoded zero amount

code/packages/router/src/contract.ts:L137-L147

```
return ResultAsync.fromPromise(
  this.txService.sendTx(
    {
      to: this.config.chainConfig[chainId].transactionManagerAddress,
      data: encodedData,
      value: constants.Zero,
      chainId,
      from: this.signerAddress,
    },
    requestContext,
  ),
```

code/packages/router/src/contract.ts:L206-L215

```
this.txService.sendTx(
  {
    chainId,
    data: fulfillData,
    to: nxtpContractAddress,
    value: 0,
    from: this.signerAddress,
  },
  requestContext,
),
```

- `approveTokensIfNeeded` will fail when using native assets

code/packages/sdk/src/transactionManager.ts:L329-L333

```
).andThen((signerAddress) => {
  const erc20 = new Contract(
    assetId,
    ERC20.abi,
    this.signer.provider ? this.signer : this.signer.connect(config.provider),
```

Recommendation

Remove complexity by requiring ERC20 compliant wrapped native assets (e.g. `WETH` instead of native `ETH`).

4.24 Router - config file is missing the `swapPools` attribute and credentials are leaked to console in case of invalid config

Minor

Description

Node startup fails due to missing `swapPools` configuration in `config.json.example`. Confidential secrets are leaked to console in the event that the config file is invalid.

Examples

```
  yarn workspace @connext/nxtp-router dev
[app] [nodemon] 2.0.12
[app] [nodemon] to restart at any time, enter `rs`
[app] [nodemon] watching path(s): .env dist/**/* ../@connext/nxtp-txservice/dist ../@connext/nxtp-contracts/dist ../@c
[app] [nodemon] watching extensions: js,json
[app] [nodemon] starting `node --enable-source-maps ./dist/index.js | pino-pretty`
[tsc]
[tsc] 13:52:29 - Starting compilation in watch mode...
[tsc]
[tsc]
[tsc] 13:52:29 - Found 0 errors. Watching for file changes.
[app] Found configFile
[app] Invalid config: {
[app]   "mnemonic": "candy maple cake sugar pudding cream honey rich smooth crumble sweet treat",
[app]   "authUrl": "https://auth.connext.network",
[app]   "natsUrl": "nats://nats1.connext.provide.network:4222,nats://nats2.connext.provide.network:4222,nats://nats3.c
[app]   "adminToken": "blahblah",
[app]   "chainConfig": {
[app]     "4": {
[app]       "providers": [
[app]         "https://rinkeby.infura.io/v3/"
[app]       ],
[app]       "confirmations": 1,
[app]       "subgraph": "https://api.thegraph.com/subgraphs/name/connext/nxtp-rinkeby",
[app]       "transactionManagerAddress": "0x29E81453AAe28A63aE12c7ED7b3F8BC16629A4Fd",
[app]       "minGas": "100000000000000000"
[app]     },
[app]     "5": {
[app]       "providers": [
[app]         "https://goerli.infura.io/v3/"
[app]       ],
[app]       "confirmations": 1,
[app]       "subgraph": "https://api.thegraph.com/subgraphs/name/connext/nxtp-goerli",
[app]       "transactionManagerAddress": "0xbF0F4f639cDd010F38CeBEd546783BD71c9e5Ea0",
[app]       "minGas": "100000000000000000"
[app]     }
[app]   },
[app]   "logLevel": "info"
[app] }
[app] Error: must have required property 'swapPools'
[app]     at Object.getEnvConfig (code/packages/router/dist/config.js:135:15)
[app]         -> code/packages/router/src/config.ts:145:11
[app]     at Object.getConfig (code/packages/router/dist/config.js:149:30)
[app]         -> code/packages/router/src/config.ts:161:18
[app]     at Object.<anonymous> (code/packages/router/dist/index.js:19:25)
[app]         -> code/packages/router/src/index.ts:23:16
[app]     at Module._compile (internal/modules/cjs/loader.js:1063:30)
[app]     at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
[app]     at Module.load (internal/modules/cjs/loader.js:928:32)
[app]     at Function.Module._load (internal/modules/cjs/loader.js:763:16)
[app]     at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
[app]     at internal/main/run_main_module.js:17:47
```

Confidential information is only cleared in case the config file is valid but not in the event of an error

code/packages/router/src/config.ts:L143-L149

```
if (!valid) {
  console.error(`Invalid config: ${JSON.stringify(nxtpConfig, null, 2)}`);
  throw new Error(validate.errors?.map((err) => err.message).join(","));
}

console.log(JSON.stringify({ ...nxtpConfig, mnemonic: "********" }, null, 2));
return nxtpConfig;
```

Recommendation

- Provide a valid default example config. Fix integration tests.
- Always remove confidential information before logging on screen.
- Avoid providing default credentials as it is very likely that someone might end up using them. Consider asking the user to provide missing credentials on first run or autogenerate it for them.
- Note that the `adminToken` is not cleared before it is being printed to screen. If this is a credential it should be blanked out before being printed. Consider separating application-specific configuration from credentials/secrets.

5 Recommendations

5.1 Router - Logging Consistency

Description

Avoid using `console.*()` in favor of the `logger.*()` family to provide a consistent timestamped log trail. Note that `console.*` might have different buffering behavior than `logger.log` which may mix up output lines.

Examples

code/packages/router/src/index.ts:L124-L131

```
server.listen(8080, "0.0.0.0", (err, address) => {
  if (err) {
    console.error(err);
    process.exit(1);
  }
  console.log(`Server listening at ${address}`);
});
```

code/packages/router/src/contract.ts:L415-L416

```
const decoded = this.txManagerInterface.decodeFunctionResult("getRouterBalance", encodedData);
console.log("decoded: ", decoded);
```

5.2 Router - Always perform strict validation of data received from third-parties or untrusted sources

Description

For example, in `subgraph.ts` an external resource is queried to return transactions that match the router’ s ID, sending Chain, and status. An honest external party will only return items that match this filter. However, in case of the third-party misbehaving (or being breached), it might happen that entries that do not belong to this node or chain configuration are returned.

Examples

code/packages/router/src/subgraph.ts:L153-L175

```
let allSenderPrepared: GetSenderTransactionsQuery;
try {
  allSenderPrepared = await sdk.GetSenderTransactions({
    routerId: this.routerAddress.toLowerCase(),
    sendingChainId: chainId,
    status: TransactionStatus.Prepared,
  });
} catch (err) {
  this.logger.error(
    { method, methodId, error: jsonifyError(err) },
    "Error in sdk.GetSenderTransactions, aborting loop interval",
  );
  return;
}

// create list of txIds for each receiving chain
const receivingChains: Record<string, string[]> = {};
allSenderPrepared.router?.transactions.forEach(({ transactionId, receivingChainId }) => {
  if (receivingChains[receivingChainId]) {
    receivingChains[receivingChainId].push(transactionId);
  } else {
    receivingChains[receivingChainId] = [transactionId];
  }
```

Recommendation

It is recommended to implement a defense-in-depth approach always validating inputs that come from third-parties or untrusted sources. Especially because the resources spent on performing the checks are negligible and significantly reduce the risk posed by third-party data providers.

5.3 FulfillInterpreter - `ReentrancyGuard` can be removed Pending

Resolution

The `nonReentrant` modifier has been removed from `FulfillInterpreter.execute`, but the import of and inheritance from `ReentrancyGuard` are still present. These are not needed anymore and should be removed too.

Description and Recommendation

The `FulfillInterpreter` has its own reentrancy protection, separate from the `TransactionManager`'s. However, the only external state-changing function in this contract, `execute`, has not only a `nonReentrant` modifier but also an `onlyTransactionManager` modifier that ensures this function can only be called from the `TransactionManager` instance to which this `FulfillInterpreter` instance belongs.

code/packages/contracts/contracts/interpreters/FulfillInterpreter.sol:L22-L28

```
/**
 * @notice Errors if the sender is not the transaction manager
 */
modifier onlyTransactionManager {
  require(msg.sender == _transactionManager, "#OTM:027");
  _;
}
```

code/packages/contracts/contracts/interpreters/FulfillInterpreter.sol:L54-L61

```
function execute(
  bytes32 transactionId,
  address payable callTo,
  address assetId,
  address payable fallbackAddress,
  uint256 amount,
  bytes calldata callData
) override external payable nonReentrant onlyTransactionManager {
```

Consequently, if the `TransactionManager` contract can't be reentered, the `FulfillInterpreter` is *automatically* protected against reentrancy. Hence, if issue 4.9 is fixed, the reentrancy guard can be removed from `FulfillInterpreter`.

5.4 FulfillInterpreter - `_transactionManager` state variable can be immutable Fixed

Resolution

This recommendation has been implemented.

Description and Recommendation

The `_transactionManager` state variable in the `FulfillInterpreter` is set in the constructor and never changed afterward. Hence, it can be `immutable`.

code/packages/contracts/contracts/interpreters/FulfillInterpreter.sol:L16-L20

```
address private _transactionManager;

constructor(address transactionManager) {
  _transactionManager = transactionManager;
}
```

5.5 TransactionManager - Risk mitigation for `addLiquidity` Fixed

Resolution

This recommendation has been implemented.

Description and Recommendation

The `addLiquidity` function has a `router` parameter to specify the beneficiary. Compared to just using `msg.sender` as beneficiary, this approach provides more flexibility, but it also increases the risk of losing funds if a wrong address is supplied. Assuming the flexibility is considered important, a lightweight risk mitigation measure is to have two separate external functions: `addLiquidity` for the presumably typical case of adding liquidity for `msg.sender`, and (perhaps) `addLiquidityFor` that takes a `router` parameter like the current implementation. The latter should only be used when necessary. Of course, both functions could utilize the same internal function to implement the actual logic.

Appendix 1 - Files in Scope

This audit covered the following files:

A.1.1 Initial version

Commit hash: `494d07d3707df91658de45e2177a79adc80cf5fe`

File	SHA-1
./packages/contracts/contracts/ProposedOwnable.sol	75ef3939477c7770c52bb2ddbb4bf5f91afe899b
./packages/contracts/contracts/TransactionManager.sol	e6976510de139b65c34a252df7a3b90b0d722d2c
./packages/contracts/contracts/interfaces/IERC20Minimal.sol	c7b725e9217869c3b0d4e9ab323a7d21492f3e17
./packages/contracts/contracts/interfaces/ITransactionManager.sol	71ebc72bdf89bce7122976a02d99809a52104e1c
./packages/contracts/contracts/interfaces/IFulfillInterpreter.sol	2f654a93a73dbdd5c20d5f4056c97ab59a58b175
./packages/contracts/contracts/interpreters/FulfillInterpreter.sol	1f570ca9204ac04c1f31710912dbc9ae873c1289
./packages/contracts/contracts/libraries/Asset.sol	4a80eb9afdd818763f5a51285b543eca00b9f209
./packages/contracts/contracts/libraries/WadRayMath.sol	9ec25c9afc02bfc0d17ab2db96a140f7c5f28b13

A.1.2 Revised version

Commit hash: `0656436d654cfe0313fa3c2bbc81aa86232ade16`

File Name	SHA-1 Hash
./packages/contracts/contracts/ProposedOwnable.sol	6d5cf96d344136b8e20ddee3894379e9cfc1a840
./packages/contracts/contracts/TransactionManager.sol	30d57ba8b435670cc879948c47a4e7dba727a21b
./packages/contracts/contracts/interfaces/IERC20Minimal.sol	c7b725e9217869c3b0d4e9ab323a7d21492f3e17
./packages/contracts/contracts/interfaces/ITransactionManager.sol	ced3ae11a7c408c2ad5a4f884ff9436419865905
./packages/contracts/contracts/interfaces/IFulfillInterpreter.sol	8e8b9f2d948fa57c0b538dd430de7decaddad94f
./packages/contracts/contracts/interpreters/FulfillInterpreter.sol	2215c41318c66450b814719717926ea5a3a48ced
./packages/contracts/contracts/lib/LibAsset.sol	5f81a92f90e8858c146174b584def6d643cb4738

Appendix 2 - Document Change Log

Version	Date	Description
1.0	2021-08-06	Report for initial code version (`494d07d3`)
1.1	2021-09-06	Updated report for revised code version (`0656436d`)
1.2	2021-12-02	Publicized report for revised code version (`0656436d`); added two unverified fixes

Appendix 3 - Disclosure

ConsenSys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.