Connext
Findings & Analysis Report

2021-08-30

Overview

ABOUT C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code 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 code contest outlined in this document, C4 conducted an analysis of Connext smart contract system written in Solidity. The code contest took place between July 8 — July 11, 2021.

WARDENS

9 Wardens contributed reports to the Connext code contest:

- 0xRajeev
- paulius.eth
- cmichel
- shw
- gpersoon
- 0xsanson
- s1m0
- hrkrshnn
- GalloDaSballo
- greiart

This contest was judged by ghoul.sol.

Final report assembled by moneylegobatman and ninek.

Summary

The C4 analysis yielded an aggregated total of 18 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 5 received a risk rating in the category of HIGH severity, 2 received a risk rating in the category of MEDIUM severity, and 11 received a risk rating in the category of LOW severity.

C4 analysis also identified 18 non-critical recommendations.

Scope

The code under review can be found within the C4 Connext code contest repository is comprised of 6 smart contracts written in the Solidity programming language.

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.

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 (5)

[H-01] ANYONE CAN ARBITRARILY ADD ROUTER LIQUIDITY

Submitted by 0xRajeev, also found by cmichel and pauliax

The `addLiquidity()` function takes a router address parameter, whose liquidity is increased (instead of assuming that `router` == `msg.sender` like is done on `removeLiquidity()`) on this contract/chain, by transferring the fund amount from router address to this contract if `assetID` != 0 (i.e. ERC20 tokens). However, anyone can call this function on the router's behalf. For `assetID` == 0, the Ether transfer via `msg.value` comes from `msg.sender` and hence is assumed to be the router itself.

The impact is that this will allow anyone to call this function and arbitrarily move ERC20 tokens from router address to this contract, assuming router has given max

approval to this contract and has `assetID` amount available for transfer. While the router can always remove the liquidity if it doesn't want to maintain that level of liquidity, this lack of access control or flexibility for a relayer to add liquidity on router's behalf, may unnecessarily (and without authorization) increase the router's exposure to protocol risk to more than it desires. See `TransactionManager.sol` #L88-L98. See also, use of `msg.sender` in `removeLiquidity` (#L88-L98).

Recommend considering the use of `msg.sender` in `addLiquidity()` or evaluate this risk otherwise.

LayneHaber (Connext) acknowledged:

> The bigger issue here is the typo here, if we use the funds from `msg.sender` that means that people are donating funds to the router.
>
> We will fix the `msg.sender`, but allow `addLiquidity` to be callable by anyone

ghoul-sol (Judge) commented:

> This is high risk because funds can be send to the wrong address.

[H-02] ACTIVETRANSACTIONBLOCKS ARE VULNERABLE TO DDOS ATTACKS

Submitted by gpersoon, also found by pauliax and shw

There is a potential issue in function `removeUserActiveBlocks` and the for loop inside it. I assume you are aware of block gas limits (they may be less relevant on other chains but still needs to be accounted for), so as there is no limit for `activeTransactionBlocks`, it may grow so large that the for loop may never finish. You should consider introducing an upper limit for `activeTransactionBlocks`. Also, a malicious actor may block any account (DDOS) by just calling `prepare` again and again with 0 amount acting as a router. This will push `activeTransactionBlocks` to the specified user until it is no longer possible to remove them from the array.

This is also a gas issue, as function `removeUserActiveBlocks` iterating and assigning large dynamic arrays is very gas-consuming. Consider optimizing the algorithm, e.g. finding the first occurrence, then swap it with the last item, pop the array, and break. Or maybe even using an `EnumerableMap`, so you can find and remove elements in `O(1)`.

The best solution depends on what the usual number of `activeTransactionBlocks` is. If it is expected to be low (e.g. less than 5), then the current approach will work. But with larger arrays, I expect `EnumerableMap` would be more efficient.

Because an upper limit will not fully mitigate this issue, as a malicious actor can still DDOS the user by pushing useless txs until this limit is reached and a valid router may not be able to submit new txs. And, because you need to improve both the security *and* performance of `removeUserActiveBlocks`; `EnumerableMap` may be a go-to solution.

LayneHaber (Connext) confirmed:

> https://github.com/connext/nxtp/pull/24

[H-03] ROUTER LIQUIDITY ON RECEIVING CHAIN CAN BE DOUBLE-DIPPED BY THE USER

Submitted by 0xRajeev, also found by cmichel, gpersoon, pauliax, s1m0 and shw

During `fulfill()` on the receiving chain, if the user has set up an external contract at `txData.callTo`, the catch blocks for both `IFulfillHelper.addFunds()` and `IFulfillHelper.excute()` perform `transferAsset` to the predetermined fallback address `txData.receivingAddress`.

If `addFunds()` has reverted earlier, `toSend` amount would already have been transferred to the `receivingAddress`. If `execute()` also fails, it is again transferred.

Scenario: User sets up receiver chain `txData.callTo` contract such that both `addFunds()` and `execute()` calls revert. That will let him get twice the `toSend` amount credited to the `receivingAddress`. So effectively, Alice locks 100 `tokenAs` on chain A, and can get 200 `tokenAs` (or twice the amount of any token she is supposed to get on chain B from the router), minus relayer fee, on chain B. Router liquidity is double-dipped by Alice and router loses funds. See `TransactionManager.sol` L395-L409 and L413-L428.

Recommend that the second catch block for `execute()` should likely not have the `transferAsset()` call. It seems like a copy-and-paste bug unless there is some reason that is outside the specified scope and documentation for this contest.

LayneHaber (Connext) confirmed and patched:

https://github.com/connext/nxtp/pull/39

[H-04] EXPIRED TRANSFERS WILL LOCK USER FUNDS ON THE SENDING CHAIN

Submitted by 0xRajeev

The cancelling relayer is being paid in `receivingAssetId` on the `sendingChain` instead of in `sendingAssetID`. If the user relies on a relayer to cancel transactions, and that `receivingAssetId` asset does not exist on the sending chain (assuming only `sendingAssetID` on the sending chain and `receivingAssetId` on the receiving chain are assured to be valid and present), then the cancel transaction from the relayer will always revert and user's funds will remain locked on the sending chain.

The impact is that expired transfers can never be cancelled and user funds will be locked forever if user relies on a relayer.

Recommend changing `receivingAssetId` to `sendingAssetId` in `transferAsset()` on `TransactionManager.sol` L514.

LayneHaber (Connext) confirmed and patched:

https://github.com/connext/nxtp/pull/25

[H-05] APPROVAL IS NOT RESET IF THE CALL TO IFULFILLHELPER FAILS

Submitted by pauliax, also found by 0xsanson, cmichel and shw

The function `fulfill` first approves the `callTo` to transfer an amount of `toSend` tokens and tries to call `IFulfillHelper`, but if the call fails, it transfers these assets directly. However, in such case the approval is not reset, so a malicous `callTo` can pull these tokens later:

```
    // First, approve the funds to the helper if needed
        if (!LibAsset.isEther(txData.receivingAssetId) && toSend > 0) {
          require(LibERC20.approve(txData.receivingAssetId, txData.callTo, to
        }

        // Next, call `addFunds` on the helper. Helpers should internally
        // track funds to make sure no one user is able to take all funds
```

```
        // track funds to make sure no one user is able to take all funds
        // for tx
        if (toSend > 0) {
          try
            IFulfillHelper(txData.callTo).addFunds{ value: LibAsset.isEther(t
              txData.user,
              txData.transactionId,
              txData.receivingAssetId,
              toSend
            )
          {} catch {
            // Regardless of error within the callData execution, send funds
            // to the predetermined fallback address
            require(
              LibAsset.transferAsset(txData.receivingAssetId, payable(txData.
              "fulfill: TRANSFER_FAILED"
            );
          }
        }
```

Tuesday, August 10, 2021 Recommend that `approval` should be placed inside the try/catch block or `approval` needs to be reset if the call fails.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/39

Medium Risk Findings (2)

[M-01] SIGNATURES USE ONLY TX ID INSTEAD OF ENTIRE DIGEST

Submitted by cmichel

The signature check in `recoverFulfillSignature()` only uses transaction ID (along with the relayer fee) which can be accidentally reused by the user, in which case the older signatures with the older relayer fees can be replayed. The signature should be on the entire digest `hashInvariantTransactionData(txData)` as indicated in the comment on L306.

The impact is that, If the user signatures are indeed on the digest as indicated by the comment, the signature/address check in `fulfill()` will fail. If not, they may be

accidentally/intentionally replayed with same transaction ID, which also appears to be an outstanding question as indicated by the comment on L12.

`recoverCancelSignature()` similarly uses only tx ID.

Unless there is a good reason not to, it is safer to include `hashInvariantTransactionData(txData)` in signatures so that they cannot be replayed with different txData (but same tx ID) whose `preparedBlockNumber` is > 0.

Recommend evaluating if the signature should contain only tx ID, or the entire digest, and then changing the logic appropriately.

LayneHaber (Connext) acknowledged:

> User should be able to break up large transfers across multiple routers using the same `transactionId` to keep the transaction unlocking atomic. For example, say I want to transfer $100K, but there are only 8 routers who each have $60K available. I should be able to break up the single transaction into $20K transactions split across 5 of the routers. When unlocking this, I should only need to broadcast a single signature, so all of the transactions can be unlocked simultaneously.

ghoul-sol (Judge) commented:

> Bumping to medium risk as replay attack can have significant consequences

[M-02] MALICIOUS ROUTER CAN BLOCK CROSS-CHAIN-TRANSFERS

Submitted by 0xRajeev, also found by cmichel and shw

The agreement between the `user` and the `router` seems to already happen off-chain because all the fields are required for the initial `In variantTransactionData` call already. A router could pretend to take on a user's cross-chain transfer, the user sends their `prepare` transaction, locking up funds on the sending chain. But then the `router` simply doesn't respond or responds with a `prepare` transaction of `amount=0`.

The user's funds are then locked for the entire expiry time, whereas the router does not have to lock up anything as the amount is 0, even no gas if they simply don't respond.

In this way, a router can bid on everything off-chain without a penalty, and take down everyone that accepts the bid.

Recommend that maybe there could be a penalty mechanism for non-responsive routers that agreed off-chain, slashing part of their added liquidity. Could also be that the bid signature already helps with this, but I'm not sure how it works as the off-chain part is not part of the repo.

LayneHaber (Connext) acknowledged:

> This is true, and we are building penalty mechanisms outside of these contracts. For now we are considering adding in a permissioned launch, see #49

Low Risk Findings (11)

[L-01] LACK OF GUARDED LAUNCH APPROACH MAY BE RISKY

Submitted by 0xRajeev, also found by pauliax

The protocol appears to allow arbitrary assets, amounts and routers/users without an initial time-bounded whitelist of assets/routers/users or upper bounds on amounts. Also, there is no pause/unpause functionality. While this lack of ownership and control makes it completely permission-less, it is a risky design because if there are latent protocol vulnerabilities there is no fallback option. See Derisking DeFi Guarded Assets.

Recommend considering an initial guarded launch approach to owner-based whitelisting asset types, router/recipient addresses, amount thresholds, and adding a pause/unpause functionality for emergency handling. The design should be able to make this owner configurable, where the owner can renounce ownership at a later point when the protocol operation is sufficiently time-tested and deemed stable/safe.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/40

[L-02] DEFLATIONARY AND FEE-ON-TRANSFER TOKENS ARE NOT CORRECTLY ACCOUNTED

Submitted by shw, also found by 0xRajeev, pauliax and cmichel

When a router adds liquidity to the `TransactionManager`, the manager does not correctly handle the received amount if the transferred token is a deflationary or fee-on-transfer token. The actual received amount is less than what is recorded in the `routerBalances` variable. See `TransactionManager.sol` #L97 and #L101.

Recommend getting the received token amount by calculating the difference of token balance before and after the transfer, for example:

```
uint256 balanceBefore = getOwnBalance(assetId);
require(LibERC20.transferFrom(assetId, router, address(this), amount, "addLic
uint256 receivedAmount = getOwnBalance(assetId) - balanceBefore;

// Update the router balances
routerBalances[router][assetId] += receivedAmount;
```

- LayneHaber (Connext) confirmed

ghoul-sol (Judge) commented:

> While correct, this is a low risk. Number of DeFi protocols are incompatible with “exotic” tokens and it’s a user responsibility to know this. Perfect example is rebase tokens and UniswapV2.

[L-03] MISSING ZERO-ADDRESS CHECKS

Submitted by 0xRajeev, also found by pauliax

Zero-address checks are in general a best-practice. However, `addLiquidity()` and `removeLiquidity()` are missing zero-address checks on router and recipient addresses respectively.

`addLiquidity()` on Eth transfers will update the zero index balance and get logged as such in the event without the amount getting accounted for the correct router.

For ERC20 assets, `token.transfer()` generally implements this check but the Eth transfer using `transferEth()` does not have this check and calls `addr.call(value)`, which will lead to burning in the case of `removeLiquidity()`.

The checks may be more important because `assetID` is 0 for Eth. So a router may accidentally use 0 values for both `assetID` and router/recipient.

There is also a missing zero-address check on `sendingChainFallback` which is relevant for Eth transfers in `cancel()`. The comment on L178 indicates the need for this but the following check on L179 ends up checking `receivingAddress` instead (which is also necessary). See issue page for referenced code.

Recommend adding zero-address checks.

sanchaymittal (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/32

[L-04] AN ATTACKER CAN FRONT-RUN A USER'S PREPARE() TX ON SENDING CHAIN TO CAUSE DOS BY GRIEFING

Submitted by 0xRajeev, also found by cmichel

The `prepare()` function hashes the invariantData parameter data to check the mapping entry is 0 for that digest as a measure to prevent duplicate `prepare()`s. However, an attacker can abuse this check to front-run a targeted victim's prepare Tx with the same parameters and with some dust amount to prevent the user's actual prepare Tx from succeeding.

The impact of this the potential griefing attack vector if user address is not `msg.sender`. This is with the assumption that relayers are only relevant on the receiving side where the user may not have the `receivingAssetId` i.e. no reason for `msg.sender` of `prepare()` to be the relayer and not the user.

Recommend adding `msg.sender` == `invariantData.user` check on sending chain side similar to the check for router address on the receiving side.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/42

LayneHaber (Connext) acknowledged:

Reverted the changes, without tracking the active blocks this DOS vector still exists but presents no risk to locked funds. It is also unclear what the attacker would gain from this, and would cost them gas funds.

Could add a signature, but comes with some UX drawbacks that seem more important than removing the DOS vector

[L-05] TXDATA.EXPIRY = BLOCK.TIMESTAMP

Submitted by pauliax, also found by 0xRajeev and shw

The function `fulfill` treats `txData.expiry` = `block.timestamp` as expired tx:

```
    // Make sure the expiry has not elapsed
    require(txData.expiry > block.timestamp, "fulfill: EXPIRED");
```

However, function `cancel` has an inclusive check for the same condition:

```
    if (txData.expiry >= block.timestamp) {
    // Timeout has not expired and tx may only be cancelled by router
```

Recommend unifying that to make the code coherent. Probably `txData.expiry` = `block.timestamp` should be treated as expired everywhere.

sanchaymittal (Connext) confirmed and patched:

https://github.com/connext/nxtp/pull/28

[L-06] UNSAFE APPROVE

Submitted by cmichel

Some ERC20 tokens like USDT require resetting the approval to 0 first before being able to reset it to another value. (See Line 201) The `LIibERC20.approve` function does not do this - unlike OpenZeppelin's `safeApprove` implementation.

The impact of this, is that repeated USDT cross-chain transfers to the same user on receiving chain = ETH mainnet can fail due to this line not resetting the approval to zero first:

```
require(LibERC20.approve(txData.receivingAssetId, txData.callTo, toSend), "f
```

Recommend that `LiibERC20.approve` should do two `approve` calls, one setting it to `0` first, then the real one. Check OpenZeppelin's `safeApprove`.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/22

[L-07] ROUTER NEEDS TO DECREASE EXPIRY BY A SIGNIFICANT BUFFER

Submitted by cmichel

The user's `fulfill` signature on the receiving chain is at the same time used by the router as a way to claim their amount on the sending chain. If the sending chain's `expiry` date has passed, the user can cancel this side of the transfer and claim back their deposit before the router can claim it. Therefore, the comment that the receiving chain's expiry needs to be decreased is correct:

> // expiry should be decremented to ensure the router has time to complete the sender-side transaction after the user completes the receiver-side transactoin.

However, this is not enforced and if a wrong expiry date is chosen by the router, or the sender congests the network long enough such that the router's `fulfill` transaction does not get through, the router loses their claim and the user gets a free cross-chain transfer.

It would be possible to enforce that `receivingSide.expiry + buffer < sendingSide.expiry` if the original expiry was part of the invariant data. This would programmatically avoid errors like the ones mentioned. (Assuming all supported chains use the same way to measure time / use UNIX timestamps.)

LayneHaber (Connext) acknowledged:

> This is true, but the router is definitely incentivized to do this correctly. Adding this would also require adding an additional `{MINIMUM/MAXIMUM}_BUFFER`, and increases the complexity of the contracts for relatively minimal benefit

[L-08] WRAPCALL WITH WEIRD ERC20 CONTRACTS

Submitted by gpersoon

The function `wrapCall` is not completely safe for all possible ERC20 contracts.

If the `returnData.length` is larger than 1, the "`abi.decode(returnData, (bool));`" will fail. Which means the interactions with that ERC20 contract will fail. Although this is unlikely, it is easy to protect against it.

```
// https://github.com/code-423n4/2021-07-connext/blob/main/contracts/lib/Lib
    function wrapCall(address assetId, bytes memory callData) internal returr
        ...
        (bool success, bytes memory returnData) = assetId.call(callData);
        LibUtils.revertIfCallFailed(success, returnData);
        return returnData.length == 0 || abi.decode(returnData, (bool));
    }
```

Recommend changing

```
return returnData.length == 0 || abi.decode(returnData, (bool));
```

to:

```
return (returnData.length == 0) || (returnData.length == 1 && abi.decode(ret
```

LayneHaber (Connext) confirmed and patched:

https://github.com/connext/nxtp/pull/22

[L-09] MAX_TIMEOUT

Submitted by pauliax

There is a `MIN_TIMEOUT` for the expiry, but I think you should also introduce a `MAX_TIMEOUT` to avoid a scenario when, for example, expiry is set far in the future (e.g. 100 years) and one malicious side does not agree to fulfill or cancel the tx, so the other side then has to wait and leave the funds locked for 100 years or so.

Recommend introducing a reasonable `MAX_TIMEOUT`.

sanchaymittal (Connext) confirmed and patched:

https://github.com/connext/nxtp/pull/30

[L-10] UNCHANGEABLE CHAINID INFORMATION

Submitted by shw

The `chainId` information included in the `TransactionManager` is immutable, i.e., it could not change after the contract is deployed. However, if a hard fork happens in the future, the contract would become invalid on one of the forked chains because the chain ID has changed. See `TransactionManager.sol` L73 and L79.

Recommend adding a function that allows the admin to set the `chainId` variable if a hard fork happens.

LayneHaber (Connext) acknowledged:

> This is a potential issue in the case of a hard fork, but we will not address it for the following reasons:
>
> 1. It is reasonable to assume the participants will want their funds to remain consistent on the canonical chain, which should keep the same `chainId` in the event of a fork
> 2. Creating an admin function to reset the `chainId` gives admins a huge amount of power over the system itself
> 3. There would still be a race between hard fork activation and updating the `chainId` that could result in unpredictable transaction behavior
>
> Instead, the course of action is to redeploy the contracts with the correct `chainId`.

[L-11] RELAYER TXS CAN BE FRONT-RUNNED

Submitted by pauliax

There is no relayer address param, only `relayerFee`, so technically anyone can front-run a profitable tx. The scenario might be as follows: A relayer submits a tx. A frontrunner sees it in the mempool and calculates that `relayerFee` is profitable enough (maybe even insta sell the `relayerFee` on AMM for the native asset) so he

copies and submits the same tx but with a higher gas price. A frontrunner's tx gets through and a relayer's tx is reverted afterward. So basically a relayer will experience only losses in such a case.

Recommend consider introducing relayer address param or reducing the probability of this scenario in any other meaningful way (e.g. blacklist front-runners).

LayneHaber (Connext) acknowledged:

> This does technically introduce some frontrun-ability for the relayer fee on the onchain transactions, but relegating the responsibility to a single relayer within the network could compromise the overall network security.
>
> Consider the following case where a relayer is selected:
>
> 1. User has a `fulfill` transaction they would like to be submitted on the receiving chain
> 2. User selects a `relayer` who will submit a tx for a fee
> 3. User sends the `relayer` the transaction data to submit the tx, including the `signature` on the receiving chain
> 4. The `relayer` can see the `router` on the transaction, and they collude to submit the signature on the sending chain, wait to cancel the transaction on the receiving chain, and split the profits.
>
> While the relayer fees are frontrunnable, and this will drive up the costs of the relayer fees for all users, switching away from this pattern will force an "all honest relayers" assumption instead of a "one honest relayer" assumption.

ghoul-sol (Judge) commented:

> Making this a low risk as the front running doesn't affect users and it actually forces the whole system to use the most optimal fees.

Non-Critical Findings

[N-01] MISSING @PARAM IN FULFILL NATSPEC

Submitted by 0xsanson

The current implementation of NatSpec of `fulfill` function lacks @param `callData` in `TransactionManager.sol` L302.

Recommend adding @param `callData`.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/38

[N-02] REFACTOR: REUSE SAME CODE FOR HASHVARIANTTRANSACTIONDATA WITH TXDATA AND WHEN PREPAREDBLOCKNUMBER IS 0

Submitted by GalloDaSballo

The code uses `hashVariantTransactionData` to verify the hash of the `VariantTransactionData` It also uses

```
    variantTransactionData[digest] = keccak256(abi.encode(VariantTransaction
      amount: txData.amount,
      expiry: txData.expiry,
      preparedBlockNumber: 0
    }));
```

To generate `VariantTransactionData` with `preparedBlockNumber` set to 0

A simple refactoring of:

```
  function hashVariantTransactionData(TransactionData calldata txData) inter
    return hashVariantTransaction(txData.amount, txData.expiry, txData.prepar
  }

  function hashVariantTransaction(uint256 amount, uint256 expiry, uint256 pre
    return keccak256(abi.encode(VariantTransactionData({
      amount: amount,
      expiry: expiry,
      preparedBlockNumber: preparedBlockNumber
    })));
  }
```

This would allow to further steamline the code from

```
    variantTransactionData[digest] = keccak256(abi.encode(VariantTransaction
      amount: txData.amount,
      expiry: txData.expiry,
      preparedBlockNumber: 0
    }));
```

to

```
    variantTransactionData[digest] = hashVariantTransaction(txData.amount, t
```

This has no particular benefit beside making all code related to Variant Data consistent.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/37

[N-03] DON'T USE ASSEMBLY

Submitted by gpersoon

The function `revertIfCallFailed` of `LibUtils.sol` uses "assembly" to log error information in a revert situation. In the latest solidity version, this can be done in solidity using the "error" keyword. See: https://docs.soliditylang.org/en/latest/control-structures.html?#revert

Using pure solidity improves readability.

`LibUtils.sol` L10

```
 function revertIfCallFailed(bool success, bytes memory returnData) internal
       if (!success) {
           assembly {  revert(add(returnData, 0x20), mload(returnData))  }
       }
    }
```

Recommend using the error constructs of solidity 0.8.4+

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/22

[N-04] CODE CONSISTENCY FOR HASHVARIANTTRANSACTIONDATA()

Submitted by greiart

`hashVariantTransactionData()` should follow the same style of `hashInvariantTransactionData()` and the recover signature functions, where the payload is generated is stored in memory before hashing. Preliminary tests in remix show that it is minimally more gas efficient as well.

```
function hashVariantTransactionData(TransactionData calldata txData) interna
    VariantTransactionData memory variant = VariantTransactionData({
      amount: txData.amount,
      expiry: txData.expiry,
      preparedBlockNumber: txData.preparedBlockNumber
        });
        return keccak256(abi.encode(variant));
  }
```

Alternative View on Notion

sanchaymittal (Connext) confirmed and patched:

https://github.com/connext/nxtp/pull/27

[N-05] STYLE ISSUES

Submitted by pauliax

`ETHER_ASSETID` is a bit missleading name, I think a better name would be `NATIVE_ASSETID`: address constant `ETHER_ASSETID` = address(0);

Misleading comment (should be 'for fulfillment'):

```
// The structure of the signed data for cancellations
struct SignedFulfillData {
```

`MIN_TIMEOUT` could be expressed in days:

```
 uint256 public constant MIN_TIMEOUT = 1 days; // 24 hours
```

sanchaymittal (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/29

sanchaymittal (Connext) commented:

> @LayneHaber (Connext) I can't find the above-mentioned typos in our latest contracts on main, not sure if changing min_timeout from hours to days makes a difference.
>
> Still created the Pr for the above.

LayneHaber (Connext) commented:

> It should be in `LibAsset`, will update

[N-06] DON'T ASK FOR THE USER'S SIGNATURE WHEN MSG.SENDER == TXDATA.USER

Submitted by pauliax

I think it would make sense not to check the user's signature in `recoverCancelSignature` or `recoverFulfillSignature` if the caller is the user himself.

Recommend replacing

```
require(recoverCancelSignature(txData, relayerFee, signature) == txData.user
require(recoverFulfillSignature(txData, relayerFee, signature) == txData.user
```

with:

```
require(msg.sender == txData.user | recoverCancelSignature(txData, relayerFe
require(msg.sender == txData.user || recoverFulfillSignature(txData, relayerF
```

LayneHaber (Connext) confirmed:

> Cannot do this on `fulfill` because then users can prevent routers from fulfilling by never revealing the proper signature. But can do this on `cancel`.
>
> PR: https://github.com/connext/nxtp/pull/41

Gas Optimizations (12)

[G-01] USING ACCESS LISTS CAN SAVE GAS DUE TO EIP-2930 POST-BERLIN HARD FORK

Submitted by 0xRajeev

EIP-2929 in Berlin fork increased the gas costs of SLOADs and CALL* family opcodes, increasing them for not-accessed slots/addresses and decreasing them for accessed slots. EIP-2930 optionally supports specifying an access list (in the transaction) of all slots and addresses accessed by the transaction, which reduces their gas cost upon access and prevents EIP-2929 gas cost increases from breaking contracts.

The impact of this is that, these changes may significantly impact gas usage for transactions that call functions touching many state variables or making many external calls. Specifically, `removeUserActiveBlocks()` removes an active block from the array of blocks for an user, all of which are stored in storage. Transactions for `fulfill()` and `cancel()` functions that call `removeUserActiveBlocks()` can consider using access lists for all the storage state (of user's active blocks) they touch (read + write) to reduce gas.

Recommend evaluating the feasibility of using access lists to save gas due to EIPs 2929 & 2930 post-Berlin hard fork. The tooling support is WIP.

LayneHaber (Connext) confirmed and patched:

> Removed tracking of active blocks: https://github.com/connext/nxtp/pull/24

[G-02] CACHE STORAGE VARIABLES TO LOCAL VARIABLES TO SAVE GAS

Submitted by shw, also found by 0xRajeev

In general, if a state variable is read more than once, caching its value to a local variable and reusing it will save gas since a storage read spends more gas than a memory write plus a memory read.

Recommend rewriting `TransactionManager.sol` #L122-L125 as follows:

```
uint256 balance = routerBalances[msg.sender][assetId];
require(balance >= amount, "removeLiquidity: INSUFFICIENT_FUNDS");

// Update router balances
routerBalances[msg.sender][assetId] = balance - amount;
```

And rewriting `TransactionManager.sol` L254-L260 as follows:

```
uint256 balance = routerBalances[invariantData.router][invariantData.receivi
require(
    balance >= amount,
    "prepare: INSUFFICIENT_LIQUIDITY"
);

// Decrement the router liquidity
routerBalances[invariantData.router][invariantData.receivingAssetId] = balanc
```

sanchaymittal (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/34

[G-03] CHECKING BEFORE EXTERNAL LIBRARY CALL CAN SAVE 2600 GAS

Submitted by 0xRajeev, also found by pauliax

EIP-2929 in Berlin fork increased the gas costs of CALL* family opcodes to 2600. Making a `delegatecall` to a library function therefore costs 2600. `LibUtils.revertIfCallFailed()` reverts and passes on the revert string if the boolean argument is false. Instead, moving the checking of the boolean to the caller avoids the library call when the boolean is true, which is likely the case most of the time.

Recommend removing the boolean parameter from `revertIfCallFailed()`, and move the conditional check logic to the call sites.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/22

[G-04] CONSOLIDATING LIBRARY FUNCTIONS CAN SAVE GAS BY PREVENTING EXTERNAL CALLS

Submitted by 0xRajeev

While code modularity is generally a good practice and creating libraries of functions commonly used across different contracts can increase maintainability and reduce contract deployment size/cost, it comes at the increased cost of gas usage at runtime because of the external calls. EIP-2929 in Berlin fork increased the gas costs of CALL* family opcodes to 2600. Making a `delegatecall` to a library function therefore costs 2600.

The impact is that, `LibAsset.transferAsset()` call from `TransactionManager.sol` makes `LibERC20.transfer()` call for ERC20 which in turn makes another external call to `LibUtils.revertIfCallFailed()` in `wrapCall`. So an ERC20 transfer effectively makes 3 additional (besides the ERC20 token contract function call `assetId.call(..)` external calls -> `LibAsset` -> `LibERC20` -> `LibUtils`, which costs 2600*3 = 7800 gas.

Combining these functions into a single library or making them all internal to `TransactionManager.sol` can convert these `delegatecall`s into JMPs to save gas. See issue page for referenced code.

Recommend considering moving all the library functions internal to this contract, or to a single library, to save gas from external calls, each of which costs 2600 gas.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/22

[G-05] EVALUATE SECURITY BENEFIT VS GAS USAGE TRADE-OFF FOR USING NONREENTRANT MODIFIER ON DIFFERENT FUNCTIONS

Submitted by 0xRajeev

While it may be considered extra-safe to have a nonreentrant modifier on all functions making any external calls even though they are to trusted contracts, when functions

implement Checks-Effects-Interactions (CEI) pattern, it is helpful to evaluate the perceived security benefit vs gas usage trade-off for using nonreentrant modifier.

Functions adhering to the CEI pattern may consider not having the nonreentrant modifier which does two `SSTORES` (getting more expensive with the London fork EIP-3529) to its `_status` state variable.

Example 1: In `addLiquidity()`, by moving the updating of router balance on L101 to before the transfers from L92, the function would adhere to CEI pattern and could be evaluated to remove the nonreentrant modifier.

Example 2: `removeLiquidity()` already adheres to CEI pattern and could be evaluated to remove the nonreentrant modifier.

`prepare()` can be slightly restructured to follow CEI pattern as well. However, `fulfill()` and `cancel()` are risky with multiple external calls and its safer to leave the nonreentrant call at the expense of additional gas costs.

The impact is that, you can save gas by removing the nonreentrant modifier if function is deemed to be reentrant safe. This can save gas costs of 2 SSTORES per function call that uses this modifier: _status SSTORE from 1 to 2 costs 5000 and _status SSTORE from 2 to 1 which costs 100 (because it was already accessed) which is significant at 5100 per call post-Berlin EIP-2929. See `TransactionManager.sol` L92-L101.

Recommend evaluating security benefit vs gas usage trade-off for using nonreentrant modifier on functions that may already be reentrant safe or do not need this protection. It may indeed be safe to leave this modifier (while accepting the gas impact) if such an evaluation is tricky or depends on assumptions.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/44

[G-06] USE THE UNCHECKED KEYWORD TO SAVE GAS

Submitted by shw, also found by 0xRajeev, cmichel, greiart, and s1m0

Using the `unchecked` keyword to avoid redundant arithmetic underflow/overflow checks to save gas when an underflow/overflow cannot happen.

We can apply the `unchecked` keyword in the following lines of code since there are `require` statements before to ensure the arithmetic operations would not cause an integer underflow or overflow.

For example, change the code at line 364 to:

```
unchecked {
    uint256 toSend = txData.amount - relayerFee;
}
```

sanchaymittal (Connext) confirmed and patched:

https://github.com/connext/nxtp/pull/33

[G-07] INCREMENT IN THE LOOP CAN BE MADE UNCHECKED

Submitted by hrkrshnn

```
 import "./interfaces/IFulfillHelper.sol";
 import "./interfaces/ITransactionManager.sol";
@@ -556,6 +556,12 @@ contract TransactionManager is ReentrancyGuard, ITransac
     return activeTransactionBlocks[user];
   }

+  function unchecked_inc(uint256 i) internal pure returns (uint256) {
+      unchecked {
+          return i + 1;
+      }
+  }
+
   ////////////////////////////
   /// Private functions ///
   ////////////////////////////
@@ -570,7 +576,7 @@ contract TransactionManager is ReentrancyGuard, ITransact
     uint256[] memory updated = new uint256[](newLength);
     bool removed = false;
     uint256 updatedIdx = 0;
-    for (uint256 i; i < newLength + 1; i++) {
+    for (uint256 i; i < newLength + 1; i = unchecked_inc(i)) {
       // Handle case where there could be more than one tx added in a block
       // And only one should be removed
       if (!removed && activeTransactionBlocks[user][i] == preparedBlock) {
```

Recommend making sure that you enable the optimizer and need at least solidity 0.8.4 (i.e., the low level inliner). The old code would have an unnecessary checked addition, and potentially also, not inlined. Would save at least 20-30 gas for each loop iteration.

LayneHaber (Connext) acknowledged:

> `unchecked` references mentioned in #74 but missed the index. We will also be using `EnumerableSet` for the active block checking

[G-08] CHECKING NON-ZERO VALUE CAN AVOID AN EXTERNAL CALL TO SAVE GAS

Submitted by 0xRajeev

Checking if `toSend` > 0 before making the external library call to `LibAsset.transferAsset()` can save 2600 gas by avoiding the external call in such situations. See L375-L380 and L364.

Recommend adding `toSend` > 0 to predicate on L375 similar to check on L387.

sanchaymittal (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/31

[G-09] OPTIMIZING THE FOR LOOP

Submitted by hrkrshnn, also found by GalloDaSballo, cmichel, gpersoon and shw

The for loop can improved, here is a diff:

```
@@ -565,22 +565,26 @@ contract TransactionManager is ReentrancyGuard, ITrans
   /// @param user User who has completed a transaction
   /// @param preparedBlock The TransactionData.preparedBlockNumber to remove
   function removeUserActiveBlocks(address user, uint256 preparedBlock) inter
-    // Remove active blocks
-    uint256 newLength = activeTransactionBlocks[user].length - 1;
-    uint256[] memory updated = new uint256[](newLength);
-    bool removed = false;
-    uint256 updatedIdx = 0;
-    for (uint256 i; i < newLength + 1; i++) {
-      // Handle case where there could be more than one tx added in a block
-      // And only one should be removed
-      if (!removed && activeTransactionBlocks[user][i] == preparedBlock) {
-        removed = true;
-        continue;
-      }
-      updated[updatedIdx] = activeTransactionBlocks[user][i];
```

```
-       updatedIdx++;
+
+    uint256[] storage array = activeTransactionBlocks[user];
+    uint256 length = array.length;
+    uint256 matchIdx = type(uint).max;
+
+    for (uint256 i = 0; i < length; i++) {
+        if (array[i] == preparedBlock)
+        {
+            matchIdx = i;
+            break;
+        }
     }
-    activeTransactionBlocks[user] = updated;
+
+    if (matchIdx != type(uint256).max) {
+        for (uint256 i = matchIdx; i < length; i++) {
+            array[i] = array[i+1];
+        }
+        array.pop();
+    }
+
}
```

The other implementation creates unnecessary copies in memory. And overwrites slots unnecessary (something like `sstore(slot,sload(slot))`). The above implementation should save significant amount of gas, by avoiding both unnecessary memory and unnecessary storage writes. Please check; was written in a hurry, but the general idea should work ☺

Optimizing the loop: Try to pass the array index instead

Instead of trying to compute the index `i` where

```
activeTransactionBlocks[user][i] == blockIndex
```

One can try to already compute this off chain and pass it as the parameter. This avoids the expensive step of reading values from storage on chain, and saves a significant amount of gas. Assume that this index `i` is passed as a parameter. On chain, all you need to do is have

```
require(activeTransactionBlocks[user][i] == blockIndex)
```

After that, use the second for loop at the end in the previous example.

LayneHaber (Connext) acknowledged:

Going to replace the `activeBlocks` with open zeppelin's `EnumerableSet`

[G-10] GAS: ONLY PASS TRANSACTIONID AS PARAMETER INSTEAD OF TRANSACTIONDATA

Submitted by cmichel

Both the `recoverFulfillSignature` and `recoverCancelSignature` functions take a large `TransactionData` object as their first argument but only use the `transactionId` field of the struct. It should be more efficient to only pass `txData.transactionId` as the parameter.

LayneHaber (Connext) confirmed and patched:

> https://github.com/connext/nxtp/pull/23

[G-11] REVERT STRINGS

Submitted by hrkrshnn

Consider using custom errors instead of revert strings. Can save gas when the revert condition has been met and also during runtime.

Consider shortening revert strings to less than 32 bytes. Revert strings more than 32 bytes require at least one additional `mstore`, along with additional operations for computing memory offset, etc.

Even if you need a string to represent an error, it can usually be done in less than 32 bytes / characters.

Here are some examples of strings that can be shortened from codebase:

```
./contracts/TransactionManager.sol:96: "addLiquidity: ETH WITH ERC TRANSFER"
./contracts/TransactionManager.sol:97: "addLiquidity: ERC20 TRANSFER FAILED"
./contracts/TransactionManager.sol:122: "removeLiquidity: INSUFFICIENT_FUNDS"
```

Note that this will only decrease runtime gas when the revert condition has been met. Regardless, it will decrease deploy time gas.

sanchaymittal (Connext) confirmed and patched:

https://github.com/connext/nxtp/pull/47

[G-12] ASSIGNMENT OF VARIABLES NOT NEEDED

Submitted by s1m0

Variables on `TransactionManager.sol` L571 and L572 are being assigned their default value so it's not needed.

Recommend removing the assignments for saving a bit of gas when deploying.

LayneHaber (Connext) acknowledged:

> We are taking out the loop in favor of the `EnumerableSet` from OpenZeppelin

---

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.