SHERLOCK

Security Review For Superfluid

Public Best Efforts Audit Contest Prepared For: **Superfluid**
Lead Security Expert: **0x73696d616f**
Date Audited: **June 4 - June 11, 2025**

Introduction

Superfluid is the money streaming protocol, powering a real-time onchain economy. Start earning every second with streaming airdrops, rewards and yield.

This contest focuses on a locker system built on top of Superfluid, which is a system showcasing programmable reward distribution.

Scope

Repository: superfluid-finance/fluid

Audited Commit: 6cb9f19564f1af239d18916e68eaa2a62597497a

Final Commit: 2a59fa6c39538cb9b420833a5ca695d91c748d87

Files:

- packages/contracts/src/EPProgramManager.sol
- packages/contracts/src/FluidEPProgramManager.sol
- packages/contracts/src/FluidLocker.sol
- packages/contracts/src/FluidLockerFactory.sol
- packages/contracts/src/Fontaine.sol
- packages/contracts/src/StakingRewardController.sol
- packages/contracts/src/interfaces/IEPProgramManager.sol
- packages/contracts/src/interfaces/IStakingRewardController.sol
- packages/contracts/src/vesting/SupVesting.sol
- packages/contracts/src/vesting/SupVestingFactory.sol

Repository: superfluid-org/protocol-monorepo

Audited Commit: 413399318428a78ef869b0c79547692a85e7b61e

Final Commit: 413399318428a78ef869b0c79547692a85e7b61e

Files:

- packages/ethereum-contracts/contracts/apps/SuperTokenV1Library.sol
- packages/ethereum-contracts/contracts/utils/MacroForwarder.sol

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
2	6

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0rpse
0x73696d616f
0xb0k0
0xbakeng
0xloscar01
0xzey
37H3RN17Y2
Angry_Mustache_Man
Artur
FalseGenius
JeRRy0422
SamuelTroyDomi
algiz
globalace
illoy_sci
katz
mahdifa
molaratai
nagato
newspacexyz
redbeans
v1c7
zacwilliamson
zxriptor

Issue H-1: Staked tokens inside FluidLocker can be withdrawn without calling Unstake

Source: https://github.com/sherlock-audit/2025-06-superfluid-locker-system-judging/issues/177

Found by

0x73696d616f, 0xzey, Angry_Mustache_Man, FalseGenius, JeRRy0422, globalace, illoy_sci, katz, mahdifa, molaratai, nagato, newspacexyz, redbeans, v1c7, zacwilliamson

Summary

Missing `getAvailableBalance()` validation in `FluidLocker::provideLiquidity` allows staked tokens to be withdrawn after the 6 months tax free period, without every calling `FluidLocker::unstake`. This means the staking rewards will accumulate despite tokens are no longer in the contract, causing a loss of integrity in the staking rewards.

Root Cause

FluidLocker::provideLiquidity does not validate against `getAvailableBalance()`, allowing tokens that are currently being staked to be used for `provideLiquidity`

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Locker owner calls FluidLocker::stake to stake all available tokens inside the locker.
2. Locker owner calls FluidLocker::provideLiquidity to create a uniswap position with these staked tokens. Note that tokens are already transferred outside of the locker at this step.
3. Locker owner calls FluidLocker::withdrawLiquidity after 6 months to trigger the tax free withdraw path. The tokens used to provide liquidity are now in Locker owner's address. However, the locker still believes they are being staked.

Impact

The staking reward points indefinitely accumulate even though the staked tokens are long gone from the contract. Since there is no upper limit on the staking amount, this leads to the loss of integrity in the staker rewards pool. People who gain ”free points” will receive the vast majority shares in future staking reward distribution.

PoC

Place the following test inside `FluidLockerTest` contract, which is located in `FluidLocker.t.sol`

```
function testWithdrawWithoutUnstake()
external
virtual
{
    uint fundingAmount = 100e18;
    // Set up Alice's Locker to be functional
    // i. e. not revert due to "lack of funds", "no LP pool units", "no staker pool
↪   units", etc.
    _helperFundLocker(address(aliceLocker), fundingAmount);
    _helperLockerStake(address(bobLocker));
    _helperLockerProvideLiquidity(address(carolLocker));

    vm.startPrank(ALICE);
    aliceLocker.stake(fundingAmount); //Stake all avaialble tokens
    aliceLocker.provideLiquidity{ value: fundingAmount / (2 * 9900) }(100e18);
↪   //Provide liquidity using the staked tokens
    vm.stopPrank();
    //warp forward to tax free withdrawn time
    vm.warp(block.timestamp +
↪   FluidLocker(payable(address(aliceLocker))).TAX_FREE_WITHDRAW_DELAY());

    //alice withdraw liquidity and closes position
    uint256 positionTokenId = _nonfungiblePositionManager.tokenOfOwnerByIndex(
        address(aliceLocker),
↪   FluidLocker(payable(address(aliceLocker))).activePositionCount() - 1
    );
    (,,,,,,, uint128 positionLiquidity,,,,) =
↪   _nonfungiblePositionManager.positions(positionTokenId);
    (uint256 amount0ToRemove, uint256 amount1ToRemove) =
↪   _helperGetAmountsForLiquidity(_pool, positionLiquidity);
    vm.prank(ALICE);
    aliceLocker.withdrawLiquidity(positionTokenId, positionLiquidity,
↪   amount0ToRemove, amount1ToRemove);

    // Check that most of contract's fluid tokens are moved to Alice's address
    // Despite us never calling unstake()
    assertApproxEqAbs(fundingAmount,
```

```
        _fluidSuperToken.balanceOf(address(ALICE)),
        fundingAmount * 5 / 100 // 5% tolerance
    );
    assertApproxEqAbs(0,
        _fluidSuperToken.balanceOf(address(aliceLocker)),
        fundingAmount * 5 / 100 // 5% tolerance
    );
    // Check that aliceLocker still believes that 100e18 tokens are staked inside it
    assertEq(aliceLocker.getStakedBalance(), fundingAmount);
    // Check that getAvailableBalance reverts because locker balance is less than
↪   staked amount
    vm.expectRevert();
    aliceLocker.getAvailableBalance();
}
```

Mitigation

Validate that the amount of tokens used to provide liquidity should never exceed getAvailableBalance()

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/superfluid-finance/fluid/pull/26

Issue H-2: Pumponomics can be skipped when using FluidLocker::provideLiquidity

Source: https://github.com/sherlock-audit/2025-06-superfluid-locker-system-judging/issues/210

Found by

0x73696d616f, 0xb0k0, 0xbakeng, 0xloscar01, 37H3RN17Y2, SamuelTroyDomi, illoy_sci, katz, nagato, newspacexyz

Summary

Pumponomics can be effectively skipped since the eth amount used to create a position on uniswap could be different than the called amount. It uses all available balance inside the contract. This causes the intended buy-pressure mechanism to be skipped, and damages Superfluid's economic model.

Root Cause

Inside FluidLocker::provideLiquidity function, only the eth sent in this call are pumped. However, all available wrapped eth are used to create a position. This allows locker owners to send eth manually to the locker, not when they are calling the provideLiquidity function, effectively skipping Pumponomics.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Locker owner transfers certain amount of wrapped ethers to the locker directly by calling WETH9.transfer, they will be used as the paired asset for provideLiquidity function.
2. Call provideLiquidity function with a dust amount of eth, and a normal amount of supAmount to utilize the wrapped ether we sent in step 1. Now a position is created without Pumponomics.

Impact

Weaken the intended structural buy-pressure; causes protocol to lose potential fees earned from swaps and liquidity.

PoC

Due to the difficulty of observing the effect of the pump function (1% difference), we directly modify the FluidLocker contract for visibility by creating a new field `uint256 public ethPumped;` and place it after all other fields that already exist. To use it, we insert this line into the beginning of the _pump function `ethPumped += ethAmount;` These modification should not change the behaviour of the contract.

Now, paste the following test into FluidLockerTest contract inside FluidLocker.t.sol If needed, please import `import { IWETH9 } from "../src/token/IWETH9.sol";`

```
function testSkipPumponomics()
external
virtual
{
    uint fundingAmount = 100e18;
    // Set up Alice's Locker to be functional
    // i. e. not revert due to "lack of funds", "no LP pool units", "no staker pool
↪   units", etc.
    _helperFundLocker(address(aliceLocker), fundingAmount);
    _helperLockerStake(address(bobLocker));
    _helperLockerProvideLiquidity(address(carolLocker));

    //Alice transfer the eth into locker via a plain call
    //Then call provideLiquidity with dust amount of eth
    //Therefore the pump function only pumps the dust eth amount, but the position
↪   is created with all eth in locker
    uint ethAmount = fundingAmount / (2 * 9900);
    address weth = _nonfungiblePositionManager.WETH9();
    vm.startPrank(ALICE);
    IWETH9(weth).deposit{ value: ethAmount }();
    IWETH9(weth).transfer(address(aliceLocker), ethAmount);
    aliceLocker.provideLiquidity{ value: 100 }(fundingAmount);
    vm.stopPrank();

    // Only a dust amount of eth has been pumped
    assertEq(FluidLocker(payable(address(aliceLocker))).ethPumped(), 1);
    // However, a position is opened with basically all the eth in the locker
    assertGt(FluidLocker(payable(address(aliceLocker))).activePositionCount(), 0);
    assertApproxEqAbs(0,
        IWETH9(weth).balanceOf(address(aliceLocker)),
        fundingAmount * 5 / 100 // 5% tolerance
    );
```

```
}
```

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/superfluid-finance/fluid/pull/27

Issue M-1: User can instantly unlock most of his funds with less fee when he is unique staker/liquidityProvider

Source: https://github.com/sherlock-audit/2025-06-superfluid-locker-system-judging/issues/47

Found by

Artur, globalace, newspacexyz

Summary

User can bypass instant `unlock` fee when he is unique `staker` and `liquidityProvider`.

Root Cause

FluidLocker.unlock checks `STAKER_DISTRIBUTION_POOL` and `LP_DISTRIBUTION_POOL` have non-zero units. But if user is unique `staker` or `liquidityProvider`, distributed penalty returns to user's locker.

```
    function _instantUnlock(uint256 amountToUnlock, address recipient) internal {
        // Calculate instant unlock penalty amount
        uint256 penaltyAmount = (amountToUnlock * _INSTANT_UNLOCK_PENALTY_BP) /
↪   BP_DENOMINATOR;


        (, uint256 providerAllocation) =
↪   STAKING_REWARD_CONTROLLER.getTaxAllocation();


        // Distribute penalty to provider (connected to the LP_DISTRIBUTION_POOL)
        uint256 actualProviderDistributionAmount =
@>          FLUID.distribute(address(this), LP_DISTRIBUTION_POOL, penaltyAmount *
↪   providerAllocation / BP_DENOMINATOR); // @audit returns to locker


        // Distribute penalty to staker (connected to the STAKER_DISTRIBUTION_POOL)
        uint256 actualStakerDistributionAmount =
@>          FLUID.distribute(address(this), STAKER_DISTRIBUTION_POOL, penaltyAmount
↪   - actualProviderDistributionAmount); // @audit returns to locker


        // Transfer the leftover $FLUID to the locker owner
        FLUID.transfer(recipient, amountToUnlock - actualProviderDistributionAmount
↪   - actualStakerDistributionAmount);
```

```
        emit FluidUnlocked(0, amountToUnlock, recipient, address(0));
    }
```

This means user can call instantly `unlock` repeatedly and he can `unlock` most of his funds with less fee. (Dust of fee can be locked in contract because penalty is always exist and unlock amount is greater than `MIN_UNLOCK_AMOUNT`)

Internal Pre-conditions

User is unique `staker/liquidityProvider`.

External Pre-conditions

.

Attack Path

1. User must be unique `staker` and unique `liquidityProvider`. (stakes and provide liquidity small amount of `FLUID` token)
2. User calls `unlock` his whole available tokens repeatedly with `unlockPeriod = 0`.
3. User's locker receives his whole penalty.
4. Penalty is 80% but he can bypass fee.
5. User `unlock` his almost available funds without fee.

Impact

User can bypass penalty(80%) and unlock immediately his almost available tokens.

PoC

No response

Mitigation

Check locker is unique `staker` or `liquidityProvider` and revert it.

```
require(STAKER_DISTRIBUTION_POOL.getTotalUnits() >
 ↪  STAKER_DISTRIBUTION_POOL.getUnits(address(this)), "Unique staker");
require(LP_DISTRIBUTION_POOL.getTotalUnits() >
 ↪  LP_DISTRIBUTION_POOL.getUnits(address(this)), "Unique Liquidity Provider");
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/superfluid-finance/fluid/pull/25

Issue M-2: User can’t unlock when getTotalUnits == 0 even though tax is zero.

Source: https://github.com/sherlock-audit/2025-06-superfluid-locker-system-judging/issues/50

Found by

newspacexyz

Summary

User can't unlock when `STAKER_DISTRIBUTION_POOL.getTotalUnits() == 0` or `LP_DISTRIBUTION_POOL.getTotalUnits() == 0`. But `stakerAllocationBP` or `liquidityProviderAllocationBP` can be zero and stakerAllocationBP + liquidityProviderAllocationBP = _BP_DENOMINATOR and this case, user can't unlock.

Root Cause

When `stakerAllocationBP + liquidityProviderAllocationBP = _BP_DENOMINATOR` but `stakerAllocationBP` or `liquidityProviderAllocationBP` is zero, one part of tax is zero so `STAKER_DISTRIBUTION_POOL.getTotalUnits()` or `LP_DISTRIBUTION_POOL.getTotalUnits()` can be zero. Users doesn't `stake` or `provideLiquidity` because tax is zero so `totalUnits` can be zero. But user can't unlock because it checks only `totalUnits`, not tax ratio.

Internal Pre-conditions

1. `stakerAllocationBP + liquidityProviderAllocationBP = _BP_DENOMINATOR`
2. `stakerAllocationBP = 0 && liquidityProviderAllocationBP = _BP_DENOMINATOR` or `stakerAllocationBP = _BP_DENOMINATOR && liquidityProviderAllocationBP = 0`
3. `stakerAllocationBP = 0 && STAKER_DISTRIBUTION_POOL.getTotalUnits() == 0` (Because `staker` reward is zero)
4. `liquidityProviderAllocationBP = 0 && LP_DISTRIBUTION_POOL.getTotalUnits() == 0` (Because `liquidityProvider` reward is zero)

External Pre-conditions

.

Attack Path

.

Impact

User can't `unlock`.

PoC

No response

Mitigation

Check tax ratio.

```
+   (uint256 stakerAllocation, uint256 providerAllocation) =
 ↪  STAKING_REWARD_CONTROLLER.getTaxAllocation();
-   if (STAKER_DISTRIBUTION_POOL.getTotalUnits() == 0) {
+   if (STAKER_DISTRIBUTION_POOL.getTotalUnits() == 0 && stakerAllocation > 0) {
        revert STAKER_DISTRIBUTION_POOL_HAS_NO_UNITS();
    }

-   if (LP_DISTRIBUTION_POOL.getTotalUnits() == 0) {
+   if (LP_DISTRIBUTION_POOL.getTotalUnits() == 0 && providerAllocation > 0) {
        revert LP_DISTRIBUTION_POOL_HAS_NO_UNITS();
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/superfluid-finance/fluid/pull/28

Issue M-3: Incorrect initial deposit calculation may cause cancelProgram to revert

Source: https://github.com/sherlock-audit/2025-06-superfluid-locker-system-judging/issues/96

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

Found by

0x73696d616f, zxriptor

Summary

The `FluidEPProgramManager.cancelProgram()` function incorrectly assumes that governance parameters (minimum deposit and liquidation period) remain constant throughout a program's lifecycle. When these parameters change after program creation, the calculated `initialDeposit` for treasury return will be incorrect, potentially causing transaction reverts or wrong accounting.

Root Cause

The vulnerability lies in the cancelProgram() function, where it recalculates the initial deposit amount using current governance parameters instead of the original values from when the program was funded:

```
uint256 buffer =
    program.token.getBufferAmountByFlowRate(programDetails.fundingFlowRate +
↪   programDetails.subsidyFlowRate);
uint256 initialDeposit =
    buffer + uint96(programDetails.fundingFlowRate +
↪   programDetails.subsidyFlowRate) * EARLY_PROGRAM_END;
```

The buffer calculation depends on two governance-controlled parameters that can be changed at any time:

1. Minimum Deposit: retrieved via SUPERTOKEN_MINIMUM_DEPOSIT_KEY
2. Liquidation Period: retrieved via CFAV1_PPP_CONFIG_KEY and decoded

The buffer amount flows through this call chain:

```
cancelProgram()
  program.token.getBufferAmountByFlowRate(totalFlowRate)
      SuperTokenV1Library.getBufferAmountByFlowRate()
          cfa.getDepositRequiredForFlowRate(token, flowRate)
```

```
            gov.getConfigAsUint256(host, token, SUPERTOKEN_MINIMUM_DEPOSIT_KEY)
            gov.getConfigAsUint256(host, token, CFAV1_PPP_CONFIG_KEY)
            SuperfluidGovernanceConfigs.decodePPPConfig(pppConfig)
            _getDepositRequiredForFlowRatePure(minimumDeposit, liquidationPeriod,
        ↪  flowRate)
                max(minimumDeposit, _clipDepositNumberRoundingUp(flowRate *
            ↪  liquidationPeriod))
```

Internal Pre-conditions

1. The program needs to be cancelled

External Pre-conditions

1. Superfluid governance changes either the minimum deposit or liquidation period parameters after program funding via:
    - setSuperTokenMinimumDeposit() for minimum deposit changes
    - setPPPConfig() for liquidation period changes

Attack Path

1. Program is created and funded with `initialDeposit` calculated using current governance parameters
2. Superfluid governance changes minimum deposit or liquidation period via:

```
npx truffle exec ops-scripts/gov-set-token-min-deposit.js : {TOKEN_ADDRESS}
 ↪  {MINIMUM_DEPOSIT}
npx truffle exec ops-scripts/gov-set-3Ps-config.js : {TOKEN_ADDRESS}
 ↪  {LIQUIDATION_PERIOD} {PATRICIAN_PERIOD}
```

3. When `cancelProgram()` is called, it recalculates `initialDeposit` using new parameters
4. If parameters increased: transaction attempts to return more funds
5. If parameters decreased: excess funds remain in contract rather than sent back to treasury

Impact

When governance parameters increase after program creation, `cancelProgram()` calculates higher buffer requirement than originally deposited, returning more funds to treasury than expected which may cause transaction revert due to insufficient fund balance. This cause inability to cancel the program.

When governance parameters decrease after program creation, `cancelProgram()` calculates lower buffer requirement than originally deposited, causing excess funds to remain in the contract while treasury receives less than the original deposit amount.

PoC

No response

Mitigation

Store the original `initialDeposit` amount in the program struct and use it during cancellation:

```
struct EPProgram {
    // ... existing fields ...
    uint256 initialDeposit;
}

// In fundProgram():
programs[programId].initialDeposit = initialDeposit;

// In cancelProgram():
token.transfer(TREASURY, programs[programId].initialDeposit);
```

This ensures the exact deposited amount is returned regardless of governance parameter changes.

Issue M-4: Fluid (SUP) can be withdrawn from the Locker while the unlock flag is false

Source: https://github.com/sherlock-audit/2025-06-superfluid-locker-system-judging/issues/207

Found by

0x73696d616f, algiz, illoy_sci

Summary

The unlock flag is meant to signal that users can unlock SUP from their Locker. However, `provideLiquidity()` and `withdrawLiquidity()` are missing this flag, so users can unlock the SUP even with the unlock flag set to false.

Root Cause

https://github.com/sherlock-audit/2025-06-superfluid-locker-system/blob/main/fluid/packages/contracts/src/FluidLocker.sol#L420 https://github.com/sherlock-audit/2025-06-superfluid-locker-system/blob/main/fluid/packages/contracts/src/FluidLocker.sol#L447 Provide liquidity and withdraw liquidity are missing the unlockAvailable modifier.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Locker has the unlock flag set to false.
2. User can provide liquidity, wait the tax free period and withdraw liquidity, while the unlock flag is still false.

Impact

Users break key functionality.

PoC

None

Mitigation

Add the unlockAvailable modifier to these functions.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/superfluid-finance/fluid/pull/30

Issue M-5: Program start failure due to incorrect buffer calculation

Source: https://github.com/sherlock-audit/2025-06-superfluid-locker-system-judging/issues/233

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

Found by

0x73696d616f

Summary

The buffer to start funding is defined as `uint256 buffer = program.token.getBufferAmountByFlowRate(fundingFlowRate + subsidyFlowRate);`.

However, due to the existence of the minimum deposit, whenever one of these rates multiplied by the liquidation period is smaller than the minimum buffer, the buffer needed will be underestimated and the program funding reverts as the resulting realtime balance of the FluidEPProgramManager is 0 after flowing (due to the wrong buffer).

This happens because it distributes 2 different flows (program + tax), so each one of them may require at least a minimum deposit. However, summed together (current estimate), they may require less than 2 minimum deposits (or only one of them would require a minimum deposit, but together with the other one, it would just increase the buffer by a smaller amount).

Root Cause

In `FluidEPProgramManager:265`, buffer is calculated as if it was 1 distributed flow, but due to the minimum deposit it is not the same.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. Admin calls `startFunding()`, but one of the flow rates is too small and the total buffer required if the flow rate is approximated as one is smaller than the effective required buffer of distributing the 2 flows separately.

Impact

DoSed `FluidEPProgramManager::startFunding()`. Not only it is time sensitive, but also it won't allow the admin to start flows with all possible values, which is key functionality.

PoC

None.

Mitigation

Calculate the buffer individually for each flow rate.

Issue M-6: Locker owners can leverage low liquidity pools to bypass the tax mechanism

Source: https://github.com/sherlock-audit/2025-06-superfluid-locker-system-judging/issues/245

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

Found by

0rpse, 0x73696d616f

Summary

Locker owners can match their SUP tokens with ETH and deploy liquidity into uniswapv3 pools, after doing this for 6 months they will get their SUP without paying tax, otherwise this tax can get up to 80%. The problem is that `FluidLocker.sol::withdrawLiquidity` transfers ETH to locker owners without considering price manipulations, because of this a locker owner can manipulate a pool's price such that liquidity will mostly consist of ETH and withdraw their liquidity in ETH, without paying taxes.

Root Cause

In `FluidLocker.sol::withdrawLiquidity` there is no check against price manipulations https://github.com/sherlock-audit/2025-06-superfluid-locker-system/blob/main/fluid/packages/contracts/src/FluidLocker.sol#L447-L485

Internal Pre-conditions

N/A

External Pre-conditions

1. Pool's liquidity is ”low”

Attack Path

Assuming 1 ETH = 2000 SUP, and the initial liquidity is spread across the whole range, since this would be more resistant to price manipulation:

1. Pool's reserves consist of 25 ETH and 50_000 SUP
2. Locker owner matches this with 25 ETH and 50_000 SUP

3. Locker owner flashloans and swaps 1500 ETH for 96764 SUP, pool reserves are 1550 ETH - 3235 SUP
4. Locker owner withdraws liquidity, getting 775 ETH paid to his address and 1617.5 SUP back in the locker
5. Locker owner swaps 46764 SUP of the 96764 SUP he had in the first swap, getting 749 ETH back In the end locker owner walks away with 22.65 ETH after paying the flashloan back with 0.09% fees, and 50K SUP in his address and 1617 SUP locked.

Impact

Depending on the liquidity present in the pools, a locker owner can bypass the tax mechanism with miniscule amounts lost compared to the normal exits present in the protocol.

Mitigation

Either make sure there is enough liquidity in the pools or use TWAP oracle to see if the price has been pushed to extreme while withdrawing liquidity.

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.