SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Surge
Prepared by: Sherlock
Lead Security Expert: 0x52
Dates Audited: February 27 - March 2, 2023
Prepared on: March 26, 2023

Introduction

Surge is an immutable lending hyperstructure for everyone and every token on every chain. Permissionless, Orcaleless and Adminless.

Scope

surge-protocol-v1 @ b7cb1dc2a2dcb4bf22c765a4222d7520843187c6

- surge-protocol-v1/src/Factory.sol
- surge-protocol-v1/src/Pool.sol

[List of all contracts in scope]

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

Medium	High
9	2

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

0x52
usmannk
chaduke
shaka
GimelSec
TrungOre
bin2chen
ctf_sec
gogo
peanuts
__141345__
Bauer
cccz
KingNFT
Tricko

unforgiven
Deivitto
joestakey
Dug
ast3ros
favelanky
SovaSlava
slvDev
BTK
0xAgro
Aymen0909
ABA
bytes032
0xhacksmithh
Cryptor
RaymondFam
MalfurionWhitehat
weeeh_

Delvir0
menox
kiki_dev
ahmedovv
wzrdk3lly
Tomo
dipp
Kaiziron
Respx
0xnuel
Handle
0xAsen
Breeje
Juntao
Bobface
SunSec
bshramin
carrot

chainNue
sinh3ck
dingo
CRYP70
gandu
ck
VAD37
0Kage
0xc0ffEE
banditx0x
rvi
ak1
Chinmay
Ace-30
gryphon
saian

Issue H-1: First depositor can abuse exchange rate to steal funds from later depositors

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/125

Found by

Breeje, Ace-30, usmannk, __141345__, TrungOre, cccz, ctf_sec, gandu, rvi, Bobface, chainNue, bin2chen, RaymondFam, Cryptor, Juntao, carrot, peanuts, ak1, saian, Chinmay, banditx0x, CRYP70, ABA, 0xc0ffEE, chaduke, 0Kage, ck, sinh3ck, MalfurionWhitehat, 0xAsen, 0xhacksmithh, bytes032, GimelSec, gryphon, unforgiven, 0x52, bshramin, SunSec, dingo, VAD37

Summary

Classic issue with vaults. First depositor can deposit a single wei then donate to the vault to greatly inflate share ratio. Due to truncation when converting to shares this can be used to steal funds from later depositors.

Vulnerability Detail

See summary.

Impact

First depositor can steal funds due to truncation

Code Snippet

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L307-L343

Tool used

Solidity YouTube Tutorial

Recommendation

Either during creation of the vault or for first depositor, lock a small amount of the deposit to avoid this.

Discussion

xeious

GG. We left this one intentionally. Glad to see this many duplicates.

Issue H-2: Precision differences when calculating userCollateralRatioMantissa causes major issues for some token pairs

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/122

Found by

gogo, GimelSec, 0x52, usmannk, __141345__, Bauer, bin2chen, TrungOre, ctf_sec, peanuts

Summary

When calculating userCollateralRatioMantissa in borrow and liquidate. It divides the raw debt value (in loan token precision) by the raw collateral balance (in collateral precision). This skew is fine for a majority of tokens but will cause issues with specific token pairs, including being unable to liquidate a subset of positions no matter what.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L474

When calculating userCollateralRatioMantissa, both debt value and collateral values are left in the native precision. As a result of this certain token pairs will be completely broken because of this. Other pairs will only be partially broken and can enter state in which it's impossible to liquidate positions.

Imagine a token pair like USDC and SHIB. USDC has a token precision of 6 and SHIB has 18. If the user has a collateral balance of 100,001 SHIB (100,001e18) and a loan borrow of 1 USDC (1e6) then their userCollateralRatioMantissa will actually calculate as zero:

```
1e6 * 1e18 / 100,001e18 = 0
```

There are two issues with this. First is that a majority of these tokens simply won't work. The other issue is that because userCollateralRatioMantissa returns 0 there are states in which some debt is impossible to liquidate breaking a key invariant of the protocol.

Any token with very high or very low precision will suffer from this.

Impact

Some token pairs will always be/will become broken

Code Snippet

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L455-L498

Tool used

Solidity YouTube Tutorial

Recommendation

userCollateralRatioMantissa should be calculated using debt and collateral values normalized to 18 decimal points

Issue M-1: fund loss because calculated Interest would be 0 in getCurrentState() due to division error

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/225

Found by

unforgiven, Tricko, joestakey, Deivitto, TrungOre

Summary

function `getCurrentState()` Gets the current state of pool variables based on the current time and other functions use it to update the contract state. it calculates interest accrued for debt from the last timestamp but because of the division error in some cases the calculated interest would be 0 and it would cause borrowers to pay no interest.

Vulnerability Detail

This is part of `getCurrentState()` code that calculates interest:

```
// 2. Get the time passed since the last interest accrual
       uint _timeDelta = block.timestamp - _lastAccrueInterestTime;

       // 3. If the time passed is 0, return the current values
       if(_timeDelta == 0) return (_currentTotalSupply, _accruedFeeShares,
 ↪  _currentCollateralRatioMantissa, _currentTotalDebt);

       // 4. Calculate the supplied value
       uint _supplied = _totalDebt + _loanTokenBalance;
       // 5. Calculate the utilization
       uint _util = getUtilizationMantissa(_totalDebt, _supplied);

       // 6. Calculate the collateral ratio
       _currentCollateralRatioMantissa = getCollateralRatioMantissa(
           _util,
           _lastAccrueInterestTime,
           block.timestamp,
           _lastCollateralRatioMantissa,
           COLLATERAL_RATIO_FALL_DURATION,
           COLLATERAL_RATIO_RECOVERY_DURATION,
           MAX_COLLATERAL_RATIO_MANTISSA,
           SURGE_MANTISSA
       );

       // 7. If there is no debt, return the current values
```

```
    if(_totalDebt == 0) return (_currentTotalSupply, _accruedFeeShares,
↪  _currentCollateralRatioMantissa, _currentTotalDebt);

    // 8. Calculate the borrow rate
    uint _borrowRate = getBorrowRateMantissa(_util, SURGE_MANTISSA, MIN_RATE,
↪  SURGE_RATE, MAX_RATE);
    // 9. Calculate the interest
    uint _interest = _totalDebt * _borrowRate * _timeDelta / (365 days *
↪  1e18); // does the optimizer optimize this? or should it be a constant?
    // 10. Update the total debt
    _currentTotalDebt += _interest;
```

code should support all the ERC20 tokens and those tokens may have different decimals. also different pools may have different values for MIN_RATE, SURGE_RATE, MAX_RATE. imagine this scenario:

1. debt token is USDC and has 6 digit decimals.
2. MIN_RATE is 5% (2 * 1e16) and MAX_RATE is 10% (1e17) and in current state borrow rate is 5% (5 * 1e16)
3. timeDelta is 2 second. (two seconds passed from last accrue interest time)
4. totalDebt is 100M USDC (100 * 1e16).
5. each year has about 31M seconds (31 * 1e6).
6. now code would calculate interest as: `_totalDebt * _borrowRate * _timeDelta / (365 days * 1e18) = 100 * 1e6 * 5 * 1e16 * 2 / (31 * 1e16 * 1e18) = 5 * 2 / 31 = 0`.
7. so code would calculate 0 interest in each interactions and borrowers would pay 0 interest. the debt decimal and interest rate may be different for pools and code should support all of them.

Impact

borrowers won't pay any interest and lenders would lose funds.

Code Snippet

https://github.com/Surge-fi/surge-protocol-v1/blob/b7cb1dc2a2dcb4bf22c765a4222d7520843187c6/src/Pool.sol#L105-L156

Tool used

Manual Review

Recommendation

don't update contract state(`lastAccrueInterestTime`) when calculated interest is 0. add more decimal to total debt and save it with extra 1e18 decimals and transferring or receiving debt token convert the token amount to more decimal format or from it.

Issue M-2: transferFrom uses allowance even if spender == from

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/214

Found by

0x52

Summary

Pool#transferFrom attempts to use allowance even when spender = from. This breaks compatibility with a large number of protocol who opt to use the transferFrom method all the time (pull only) instead of using both transfer and transferFrom (push and pull). The ERC20 standard only does an allowance check when spender != from. The result of this difference will likely result in tokens becoming irreversibly stranded across different protocols.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L284-L293

The trasnferFrom method shown above always uses allowance even if spender = from.

Impact

Token won't be compatible with some protocols and will end up stranded

Code Snippet

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L284-L293

Tool used

Solidity YouTube Tutorial

Recommendation

Only use allowance when spender != from:

```
    require(to != address(0), "Pool: to cannot be address 0");
+   if (from != msg.sender) {
```

```
+       allowance[from][msg.sender] -= amount;
+   }
    balanceOf[from] -= amount;
```

Issue M-3: # [H-02] Approve and transferFrom functions of Pool tokens are subject to front-run attack.

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/154

Found by

Delvir0, RaymondFam, Cryptor, dipp, menox, kiki_dev, wzrdk3lly, ahmedovv, weeeh_, Tricko, ABA, Respx, Tomo, MalfurionWhitehat, 0xhacksmithh, bytes032, Kaiziron, Dug, Handle, ast3ros, 0xnuel

Summary

`Approve` and `transferFrom` functions of Pool tokens are subject to front-run attack because the `approve` method overwrites the current allowance regardless of whether the spender already used it or not. In case the spender spent the amonut, the `approve` function will approve a new amount.

Vulnerability Detail

The `approve` method overwrites the current allowance regardless of whether the spender already used it or not. It allows the spender to front-run and spend the amount before the new allowance is set.

Scenario:

- Alice allows Bob to transfer N of Alice's tokens (N>0) by calling the `pool.approve` method, passing the Bob's address and N as the method arguments
- After some time, Alice decides to change from N to M (M>0) the number of Alice's tokens Bob is allowed to transfer, so she calls the `pool.approve` method again, this time passing the Bob's address and M as the method arguments
- Bob notices the Alice's second transaction before it was mined and quickly sends another transaction that calls the `pool.transferFrom` method to transfer N Alice's tokens somewhere
- If the Bob's transaction will be executed before the Alice's transaction, then Bob will successfully transfer N Alice's tokens and will gain an ability to transfer another M tokens Before Alice noticed that something went wrong, Bob calls the `pool.transferFrom` method again, this time to transfer M Alice's tokens.
- So, an Alice's attempt to change the Bob's allowance from N to M (N>0 and M>0) made it possible for Bob to transfer N+M of Alice's tokens, while Alice never wanted to allow so many of her tokens to be transferred by Bob.

Impact

It can result in losing pool tokens of users when he approve pool tokens to any malicious account.

Code Snippet

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L284 https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L299

Tool used

Manual Review

Recommendation

Use `increaseAllowance` and `decreaseAllowance` instead of approve as OpenZeppelin ERC20 implementation. Please see details here:

https://forum.openzeppelin.com/t/explain-the-practical-use-of-increaseallowance-and-decreaseallowance-functions-on-erc20/15103/4

Issue M-4: Attackers may skip the collateral ratio recovery duration to inflate collateralization ratios and steal funds

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/130

Found by

usmannk

Summary

Under certain market conditions, attackers can bypass the collateralization ratio update system to instantly send the collateralization ratio from zero to maximum.

The collateralization ratio in Surge defines the amount of loan token that can be borrowed per wei of collateral token. In this way it acts as both a collateralization ratio as in other lending markets but also as an exchange rate, as the ratio is taken directly between the two tokens instead of through an intermediary such as dollars.

A Surge pool linearly decreaes the collateralization ratio to zero when it is in "Surge mode", in order to reduce demand. When the pool is not in "Surge mode", it increases the collateralization ratio similarly up to the preset maximum. However, an attacker can use a quirk in the way the utilization is calculated to instantly send a pool from zero to the maximum, greatly manipulating the exchange rate.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L216-L263

The collateralization ratio is calculated as either

`_lastCollateralRatioMantissa + timeDelta * _maxCollateralRatioMantissa / _collateralRatioRecoveryDuration` or `_lastCollateralRatioMantissa - timeDelta * _maxCollateralRatioMantissa / _collateralRatioFallDuration`

Depending on whether the contract is not, or is, in Surge mode (respectively).

Consider a pool that has been in a state that is just under Surge mode for a long time. That is `_util <= _surgeMantissa` but the two values are very close together. There have not been any interactions with the pool in a while so `timeDelta` is large. This means that the next interaction with the pool will use a collateralization ratio of 0.

An attacker can bypass this by gifting a small amount of the loan token to the contract. This will cause the utilization (`_util`) to go up on the next calculation.

Now even though the contract has not been in surge mode, on the next interaction it will think that it has been and the large `timeDelta` will be applied to a collateralization ratio increase instead of a collateralization ratio decrease.

The attacker can take advantage of the maximum collateralization ratio to borrow funds at an inflated valuation, stealing assets from the pool because they were able to skip the `_collateralRatioRecoveryDuration`.

Impact

Loss of funds from pool depositors.

Code Snippet

Tool used

Manual Review

Recommendation

When calculating utilization, only use the most recently cached values of token balances instead of using live values.

Issue M-5: Operator can cause fee shares to be minted to address(0)

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/124

Found by

favelanky, slvDev, gogo, GimelSec, 0xAgro, 0x52, BTK, Dug, SovaSlava, Aymen0909, bin2chen, ast3ros, ctf_sec

Summary

When setting the fee rate it is required that the fee recipient is NOT address(0). An operator can bypass this check by changing the fee recipient to address(0) after setting fee.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Factory.sol#L60-L65

When setting the fee it is required that if the fee != 0 then the fee recipient != address(0)

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Factory.sol#L52-L55

When setting the fee recipient there is no similar check. This means that an operator can bypass the check in setFeeMantissa by setting the fee recipient to address(0) after setting a nonzero fee value.

Impact

Operator can bypass fee recipient check

Code Snippet

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Factory.sol#L52-L55

Tool used

Solidity YouTube Tutorial

Recommendation

Implement a check similar to the one in setFeeMantissa that doesn't allow a nonzero fee when fee recipient = address(0)

Issue M-6: Fee share calculation is incorrect

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/113

Found by

GimelSec, 0x52, cccz, KingNFT

Summary

Fees are given to the feeRecipient by minting them shares. The current share calculation is incorrect and always mints too many shares the fee recipient, giving them more fees than they should get.

Vulnerability Detail

The current equation is incorrect and will give too many shares, which is demonstrated in the example below.

Example:

```
_supplied = 100
_totalSupply = 100

_interest = 10
fee = 2
```

Calculate the fee with the current equation:

```
_accuredFeeShares = fee * _totalSupply / supplied = 2 * 100 / 100 = 2
```

This yields 2 shares. Next calculate the value of the new shares:

```
2 * 110 / 102 = 2.156
```

The value of these shares yields a larger than expected fee. Using a revised equation gives the correct amount of fees:

```
_accuredFeeShares = (_totalSupply * fee) / (_supplied + _interest - fee) = 2 *
↪  100 / (100 + 10 - 2) = 1.852

1.852 * 110 / 101.852 = 2
```

This new equation yields the proper fee of 2.

Impact

Fee recipient is given more fees than intended, which results in less interest for LPs

Code Snippet

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L161-L165

Tool used

Solidity YouTube Tutorial

Recommendation

Use the modified equation shown above:

```
    uint fee = _interest * _feeMantissa / 1e18;
    // 13. Calculate the accrued fee shares
-   _accruedFeeShares = fee * _totalSupply / _supplied; // if supplied is 0, we
↪   will have returned at step 7
+   _accruedFeeShares = fee * (_totalSupply * fee) / (_supplied + _interest -
↪   fee); // if supplied is 0, we will have returned at step 7
    // 14. Update the total supply
    _currentTotalSupply += _accruedFeeShares;
```

Issue M-7: Attackers can force surge to never update the collateralization ratio

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/109

Found by

usmannk

Summary

Certain parameter choices make it feasible to block updates to the collateralization ratio. Collaterization ratio updates are calculated as `uint change = timeDelta * _maxCollateralRatioMantissa / _collateralRatioRecoveryDuration;` . However, with quick refreshes or a `_collateralRatioRecoveryDuration` that is greater than `_maxCollateralRatioMantissa`, this change may be zero every iteration.

Vulnerability Detail

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L216-L263

The `getCollateralRatioMantissa` function calculates the collateralization ratio by linearly updating along `_maxCollateralRatioMantissa / _collateralRatioRecoveryDuration`. However, these updates may be forced to zero in certain situations.

Consider a pool where the loan token is WBTC and the collateral token is DAI. Given a BTC price of $20,000 it is reasonable to only allow 1/10000 BTC to be borrowed per DAI (for a max rate of $10,000 per BTC).

The `_maxCollateralRatioMantissa` in this case would be `1e14`. In the Surge tests, a `_collateralRatioRecoveryDuration` of `1e15` is used. If an attacker does a tiny deposit of 1wei WBTC more often than once every 10 seconds, the `change` of the max collateralization ratio will always be zero no matter what the current utilization is because `(timeDelta * _maxCollateralRatioMantissa)` is less than `_collateralRatioRecoveryDuration`.

This would halt the entire adaptive pricing scheme of the Surge protocol while still allowing borrows at the current rate.

The README specifies that Surge is meant to be deployed on `DEPLOYMENT: Mainnet, Optimism, Arbitrum, Fantom, Avalanche, Polygon, BNB Chain and other EVM chains`. This exploit is especially attractive on L2s because of cheap/free execution (e.g. Optimism) and very low block times (thus low `timeDelta`).

Impact

Loss of funds for depositors as the price becomes stale and the collateralization rate, and thus pool exchange rate, of the Surge pool would no longer update.

Code Snippet

Tool used

Manual Review

Recommendation

Ensure that `_collateralRatioRecoveryDuration < _maxCollateralRatioMantissa`. This would preclude some pools from existing, but save funds from being stolen.

Discussion

hrishibhat

Given the unlikely edge case of having a pool with an edge case mentioned by the Sponsor:

> a legit pool might have recovery duration set to max uint in case lenders wouldn't want the collateral factor to ever rise back up after falling

Considering this issue as a valid medium

Issue M-8: Users can borrow all loan tokens

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/106

Found by

shaka

Summary

Utilization rate check can be bypassed depositing additional loan tokens and withdrawing them in the same transaction.

Vulnerability Detail

In the `borrow` function it is checked that the new utilization ratio will not be higher than the *surge threshold*. This threshold prevents borrowers from draining all available liquidity from the pool and also trigger the *surge state*, which lowers the collateral ratio.

A user can bypass this and borrow all available loan tokens following these steps:

- Depositing the required amount of loan tokens in order to increase the balance of the pool.
- Borrow the remaining loan tokens from the pool.
- Withdraw the loan tokens deposited in the first step.

This can be done in one transaction and the result will be a utilization rate of 100%. Even if the liquidity of the pool is high, the required loan tokens to perform the strategy can be borrowed using a flash loan.

Impact

The vulnerability allows to drain all the liquidity from the pool, which entails two problems:

- The collateral ratio starts decreasing and only stops if the utilization ratio goes back to the surge threshold.
- The suppliers will not be able to withdraw their tokens.

The vulnerability can be executed by the same or other actors every time a loan is repaid or a new deposit is done, tracking the mempool and borrowing any new amount of loan tokens available in the pool, until the collateral ratio reaches a value of zero.

A clear case with economic incentives to perform this attack would be that the collateral token drops its price at a high rate and borrow all the available loan tokens from the pool, leaving all suppliers without the chance of withdrawing their share.

Code Snippet

https://github.com/Surge-fi/surge-protocol-v1/blob/b7cb1dc2a2dcb4bf22c765a4222d7520843187c6/src/Pool.sol#L477-L478

Proof of concept

Helper contract:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.17;

import { FlashBorrower, Flashloan, IERC20Token } from "./FlashLoan.sol";
import { Pool } from "./../../src/Pool.sol";

contract Borrower is FlashBorrower {
    address public immutable owner;
    Flashloan public immutable flashLoan;
    Pool public immutable pool;
    IERC20Token public loanToken;

    constructor(Flashloan _flashLoan, Pool _pool) {
        owner = msg.sender;
        flashLoan = _flashLoan;
        pool = _pool;
        loanToken = IERC20Token(address(_pool.LOAN_TOKEN()));
    }

    function borrowAll() public returns (bool) {
        // Get current values from pool
        pool.withdraw(0);
        uint loanTokenBalance = loanToken.balanceOf(address(pool));
        loanToken.approve(address(pool), loanTokenBalance);

        // Execute flash loan
        flashLoan.execute(FlashBorrower(address(this)), loanToken,
↪   loanTokenBalance, abi.encode(loanTokenBalance));
    }

    function onFlashLoan(IERC20Token token, uint amount, bytes calldata data)
↪   public override {
        // Decode data
        (uint loanTokenBalance) = abi.decode(data, (uint));
```


SHERLOCK

```
        // Deposit tokens borrowed from flash loan, borrow all other LOAN tokens
↪   from pool and
        // withdraw the deposited tokens
        pool.deposit(amount);
        pool.borrow(loanTokenBalance);
        pool.withdraw(amount);

        // Repay the loan
        token.transfer(address(flashLoan), amount);

        // Send loan tokens to owner
        loanToken.transfer(owner, loanTokenBalance);
    }
}
```

Execution:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.17;

import "forge-std/Test.sol";
import "../src/Pool.sol";
import "../src/Factory.sol";
import "./mocks/Borrower.sol";
import "./mocks/ERC20.sol";

contract PoC is Test {
    address alice = vm.addr(0x1);
    address bob = vm.addr(0x2);
    Factory factory;
    Pool pool;
    Borrower borrower;
    Flashloan flashLoan;
    MockERC20 collateralToken;
    MockERC20 loanToken;
    uint maxCollateralRatioMantissa;
    uint surgeMantissa;
    uint collateralRatioFallDuration;
    uint collateralRatioRecoveryDuration;
    uint minRateMantissa;
    uint surgeRateMantissa;
    uint maxRateMantissa;

    function setUp() public {
        factory = new Factory(address(this), "G");
        flashLoan = new Flashloan();
        collateralToken = new MockERC20(1 ether, 18);
```

```
        collateralToken.transfer(bob, 1 ether);
        loanToken = new MockERC20(100 ether, 18);
        loanToken.transfer(alice, 1 ether);
        loanToken.transfer(address(flashLoan), 99 ether);
        maxCollateralRatioMantissa = 1e18;
        surgeMantissa = 0.8e18; // 80%
        pool = factory.deploySurgePool(IERC20(address(collateralToken)),
    ↪ IERC20(address(loanToken)), maxCollateralRatioMantissa, surgeMantissa, 1e15,
    ↪ 1e15, 0.1e18, 0.4e18, 0.6e18);
    }

    function testFailBorrowAll() external {
        // Alice deposits 1 LOAN token
        vm.startPrank(alice);
        loanToken.approve(address(pool), 1 ether);
        pool.deposit(1 ether);
        vm.stopPrank();

        // Bob tries to borrow all available loan tokens
        vm.startPrank(bob);
        collateralToken.approve(address(pool), 1 ether);
        pool.addCollateral(bob, 1 ether);
        pool.borrow(1 ether);
        vm.stopPrank();
    }

    function testBypassUtilizationRate() external {
        uint balanceBefore = loanToken.balanceOf(bob);

        // Alice deposits 1 LOAN token
        vm.startPrank(alice);
        loanToken.approve(address(pool), 1 ether);
        pool.deposit(1 ether);
        vm.stopPrank();

        // Bob tries to borrow all available loan tokens
        vm.startPrank(bob);
        collateralToken.approve(address(pool), 1 ether);
        borrower = new Borrower(flashLoan, pool);
        pool.addCollateral(address(borrower), 1 ether);
        borrower.borrowAll();
        vm.stopPrank();

        assertEq(loanToken.balanceOf(bob) - balanceBefore, 1 ether);
    }
}
```

Tool used

Manual Review

Recommendation

A possible solution would be adding a locking period for deposits of loan tokens.

Another possibility is to enforce that the utilization rate was under the surge rate also in the previous snapshot.

Discussion

xeious

Recommending medium severity because there's no direct loss of funds. We're thinking of solving this by forbidding deposits and withdrawals in a single block.

Issue M-9: A liquidator can gain not only collateral, but also can reduce his own debt!

Source: https://github.com/sherlock-audit/2023-02-surge-judging/issues/101

Found by

chaduke

Summary

A liquidator can gain not only collateral, but also can reduce his own debt. This is achieved by taking advantage of the following vulnerability of the `liquidate()`: it has a rounding down precision error and when one calls liquidate(Bob, 1), it is possible that the total debt is reduced by 1, but the debt share is 0, and thus Bob's debt shares will not be reduced. In this way, the liquidator can shift part of debt to the remaining borrowers while getting the collateral of the liquidation.

In summary, the liquidator will be able to liquidate a debtor, grab proportionately the collateral, and in addition, reduce his own debt by shifting some of his debt to the other borrowers.

Vulnerability Detail

Below, I explain the vulnerability and then show the code POC to demonstate how a liquidator can gain collateral as well as reduce his own debt!

1) The `liquidate()` function calls `tokenToShares()` at L587 to calculate the number of debt shares for the input `amount`. Note it uses a rounding-down.

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L553-L609

2) Due to rounding down, it is possible that while `amount !=0`, the returned number of debt shares could be zero!

https://github.com/sherlock-audit/2023-02-surge/blob/main/surge-protocol-v1/src/Pool.sol#L199-L204

3) In the following code POC, we show that Bob (the test account) and Alice (address(1)) both borrow 1000 loan tokens, and after one year, each of them owe 1200 loan tokens. Bob liquidates Alice's debt with 200 loan tokens. Bob gets the 200 collateral tokens (proportionately). In addition, Bob reduces his own debt from 1200 to 1100!

To run this test, one needs to change `pool.getDebtOf()` as a public function.

```
function testLiquidateSteal() external {
        uint loanTokenAmount = 12000;
        uint borrowAmount = 1000;
        uint collateralAmountA = 10000;
        uint collateralAmountB = 1400;
        MockERC20 collateralToken = new
↪   MockERC20(collateralAmountA+collateralAmountB, 18);
        MockERC20 loanToken = new MockERC20(loanTokenAmount, 18);
        Pool pool = factory.deploySurgePool(IERC20(address(collateralToken)),
↪   IERC20(address(loanToken)), 0.8e18, 0.5e18, 1e15, 1e15, 0.1e18, 0.4e18,
↪   0.6e18);
        loanToken.approve(address(pool), loanTokenAmount);
        pool.deposit(loanTokenAmount);

        // Alice borrows 1000
        collateralToken.transfer(address(1), collateralAmountB);
        vm.prank(address(1));
        collateralToken.approve(address(pool), collateralAmountB);
        vm.prank(address(1));
        pool.addCollateral(address(1), collateralAmountB);
        vm.prank(address(1));
        pool.borrow(borrowAmount);

        // Bob borrows 1000 too
        collateralToken.approve(address(pool), collateralAmountA);
        pool.addCollateral(address(this), collateralAmountA);
        pool.borrow(borrowAmount);

       // Bob's debt becomes 1200
       vm.warp(block.timestamp + 365 days);
       pool.withdraw(0);
       uint mydebt = pool.getDebtOf(pool.debtSharesBalanceOf(address(this)),
↪   pool.debtSharesSupply(), pool.lastTotalDebt());
       assertEq(mydebt, 1200);

      // Alice's debt becomes 1200
       uint address1Debt = pool.getDebtOf(pool.debtSharesBalanceOf(address(1)),
↪   pool.debtSharesSupply(), pool.lastTotalDebt());
       assertEq(address1Debt, 1200);
       assertEq(pool.lastTotalDebt(), 2399);

       uint myCollateralBeforeLiquidate =
↪   collateralToken.balanceOf(address(this));

       // liquidate 200 for Alice
       loanToken.approve(address(pool), 200);
```

```
        for(int i; i<200; i++)
            pool.liquidate(address(1), 1);

      // Alice's debt shares are NOT reduced, now Bob's debt is reduced to 1100
        uint debtShares = pool.debtSharesBalanceOf(address(1));
        assertEq(debtShares, 1000);
        assertEq(pool.lastTotalDebt(), 2199);
        address1Debt = pool.getDebtOf(pool.debtSharesBalanceOf(address(1)),
↪   pool.debtSharesSupply(), pool.lastTotalDebt());
        assertEq(address1Debt, 1100);
        mydebt = pool.getDebtOf(pool.debtSharesBalanceOf(address(this)),
↪   pool.debtSharesSupply(), pool.lastTotalDebt());
        assertEq(mydebt, 1100);

        // Bob gains the collateral as well proportionately
        uint myCollateralAfterLiquidate =
↪   collateralToken.balanceOf(address(this));
        assertEq(myCollateralAfterLiquidate-myCollateralBeforeLiquidate, 200);
    }
```

Impact

A liquidator can gain not only collateral, but also can reduce his own debt. Thus, he effectively steals funding from the pool by off-shifting his debt to the remaining borrowers.

Code Snippet

See above

Tool used

VScode

Manual Review

Recommendation

We need to double check this edge case and now allowing the liquidate() to proceed when the # of debt shares is Zero.

```
function liquidate(address borrower, uint amount) external {
       uint _loanTokenBalance = LOAN_TOKEN.balanceOf(address(this));
       (address _feeRecipient, uint _feeMantissa) = FACTORY.getFee();
       (
           uint _currentTotalSupply,
```

```
            uint _accruedFeeShares,
            uint _currentCollateralRatioMantissa,
            uint _currentTotalDebt
        ) = getCurrentState(
            _loanTokenBalance,
            _feeMantissa,
            lastCollateralRatioMantissa,
            totalSupply,
            lastAccrueInterestTime,
            lastTotalDebt
        );

        uint collateralBalance = collateralBalanceOf[borrower];
        uint _debtSharesSupply = debtSharesSupply;
        uint userDebt = getDebtOf(debtSharesBalanceOf[borrower],
 ↪  _debtSharesSupply, _currentTotalDebt);
        uint userCollateralRatioMantissa = userDebt * 1e18 / collateralBalance;
        require(userCollateralRatioMantissa > _currentCollateralRatioMantissa,
 ↪  "Pool: borrower not liquidatable");

        address _borrower = borrower; // avoid stack too deep
        uint _amount = amount; // avoid stack too deep
        uint _shares;
        uint collateralReward;
        if(_amount == type(uint).max || _amount == userDebt) {
            collateralReward = collateralBalance;
            _shares = debtSharesBalanceOf[_borrower];
            _amount = userDebt;
        } else {
            uint userInvertedCollateralRatioMantissa = collateralBalance * 1e18
 ↪  / userDebt;
            collateralReward = _amount * userInvertedCollateralRatioMantissa /
 ↪  1e18; // rounds down
            _shares = tokenToShares(_amount, _currentTotalDebt,
 ↪  _debtSharesSupply, false);
        }

+     if(_shares == 0) revert ZeroShareLiquidateNotAllowed();

        _currentTotalDebt -= _amount;

        // commit current state
        debtSharesBalanceOf[_borrower] -= _shares;
        debtSharesSupply = _debtSharesSupply - _shares;
        collateralBalanceOf[_borrower] = collateralBalance - collateralReward;
        totalSupply = _currentTotalSupply;
        lastTotalDebt = _currentTotalDebt;
        lastAccrueInterestTime = block.timestamp;
```

```
        lastCollateralRatioMantissa = _currentCollateralRatioMantissa;
        emit Liquidate(_borrower, _amount, collateralReward);
        if(_accruedFeeShares > 0) {
            address __feeRecipient = _feeRecipient; // avoid stack too deep
            balanceOf[__feeRecipient] += _accruedFeeShares;
            emit Transfer(address(0), __feeRecipient, _accruedFeeShares);
        }

        // interactions
        safeTransferFrom(LOAN_TOKEN, msg.sender, address(this), _amount);
        safeTransfer(COLLATERAL_TOKEN, msg.sender, collateralReward);
    }
```

Discussion

xeious

Low potential impact of precision loss. Recommending medium severity.