SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Flat Money
Prepared by: Sherlock
Lead Security Expert: xiaoming90
Dates Audited: April 9 - April 12, 2024
Prepared on: May 13, 2024

Introduction

The Flat Money protocol allows people to deposit Rocket Pool ETH (rETH) and mint UNIT, a decentralized delta-neutral flatcoin designed to outpace inflation. Flat Money also offers Leverage Traders the ability to deposit rETH and open rETH leveraged long positions through perpetual futures contracts.

Scope

Repository: dhedge/flatcoin-v1

Branch: master

Commit: ff4d191ac6766bc695e96657899e85446ec858b1

For the detailed scope, see the contest details.

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
3	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

xiaoming90
santipu_
bareli
Bauchibred

Issue M-1: Large amount of points can STILL be minted without any cost

Source: https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest-judging/issues/10

The protocol has acknowledged this issue.

Found by

Bauchibred, santipu_, xiaoming90

Summary

An issue was raised in the last FlatMoney audit (here) where the watsons pointed out that the points could be minted without any cost, this issue still remains, and now the attacker can prevent other users from earning points.

Vulnerability Detail

The issue raised in the last audit pointed out that an attacker could perform the following attack to mint a large number of points:

1. Deposit liquidity
2. After a short amount of time, withdraw it
3. Repeat the attack, minting a huge amount of points

This attack can still be executed to mint a large number of points at almost no cost. The mitigation that was implemented (if I'm not mistaken), consisted of a rate limit on the minted points. Even though this mitigation would reduce the profit of an attacker, it won't prevent the attacker from minting the maximum amount of points until the rate limit is reached.

Moreover, an attacker could use the rate limit to prevent other users from minting points. By constantly depositing and withdrawing liquidity, it would mint all points for himself until the rate limit is reached. Then, when other innocent users deposit liquidity or make trades, they won't receive any points.

Impact

The attacker can still mint a large amount of points, and prevent other users from receiving them.

Code Snippet

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/StableModule.sol#L93

Tool used

Manual Review

Recommendation

Reduce the amount of points earned when users withdraw liquidity or implement a withdrawal fee even if the LP is the last in the market.

Discussion

sherlock-admin2

2 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

takarez commented:

> valid; medium(1)

itsermin

There is a cost associated with withdrawing liquidity in the form of the withdrawal fee. If a user wishes to mint points and giving collateral fees to LP holders, then that's their choice. It's an exchange. These fees increase the returns for the LPs and create incentive for more deposits.

There will be no change to this functionality.

Issue M-2: Code asymmetry of `globalPositions.marginDepositedTotal`

Source: https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest-judging/issues/19

Found by

xiaoming90

Summary

Code asymmetry of `globalPositions.marginDepositedTotal` state variable could lead to an accounting error within the protocol, leading to the protocol being broken. It will also lead to a loss of assets for the LP.

Vulnerability Detail

There is a code asymmetry issue in how the `globalPositions.marginDepositedTotal` is calculated within the `FlatcoinVault.settleFundingFees` and `FlatcoinVault.updateGlobalPositionData` functions.

Within the `FlatcoinVault.settleFundingFees` function, the `globalPositions.marginDepositedTotal` is allowed to go to negative. The reason for allowing this is mentioned in the code's comment `Once the underwater position is liquidated, then the funding fees will be reverted and the total will be positive again.` in Line 238 of the `FlatcoinVault.settleFundingFees` function. During the liquidation of the underwater position, the liquidation logic will re-adjust the negative `globalPositions.marginDepositedTotal` to be positive again.

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L224

```
File: FlatcoinVault.sol
224:     function settleFundingFees() public returns (int256 _fundingFees) {
..SNIP..
234:         // Calculate the funding fees accrued to the longs.
235:         // This will be used to adjust the global margin and collateral
 ↪  amounts.
236:         _fundingFees =
 ↪  PerpMath._accruedFundingTotalByLongs(_globalPositions, unrecordedFunding);
237:
238:         // In the worst case scenario that the last position which remained
 ↪  open is underwater,
```

```
239:         // We set the margin deposited total to negative. Once the
↪  underwater position is liquidated,
240:         // then the funding fees will be reverted and the total will be
↪  positive again.
241:         _globalPositions.marginDepositedTotal =
↪  _globalPositions.marginDepositedTotal + _fundingFees;
242:
243:         _updateStableCollateralTotal(-_fundingFees);
244:     }
```

However, within the `FlatcoinVault.updateGlobalPositionData` functions, it operates differently where the `globalPositions.marginDepositedTotal` cannot become negative. If the value is negative, it will be set to zero, as seen in Line 191 of the `FlatcoinVault.updateGlobalPositionData` functions above.

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L175

```
File: FlatcoinVault.sol
175:     function updateGlobalPositionData(
176:         uint256 _price,
177:         int256 _marginDelta,
178:         int256 _additionalSizeDelta
179:     ) external onlyAuthorizedModule {
180:         // Note that technically, even the funding fees should be accounted
↪  for when computing the margin deposited total.
181:         // However, since the funding fees are settled at the same time as
↪  the global position data is updated,
182:         // we can ignore the funding fees here.
183:         int256 newMarginDepositedTotal =
↪  _globalPositions.marginDepositedTotal + _marginDelta;
184:
185:         // Check that the sum of margin of all the leverage traders is not
↪  negative.
186:         // Rounding errors shouldn't result in a negative margin deposited
↪  total given that
187:         // we are rounding down the profit loss of the position.
188:         // The margin may be negative if liquidations are not happening in
↪  a timely manner.
189:         // In such a case, the system should continue to function as normal.
190:         if (newMarginDepositedTotal < 0) {
191:             newMarginDepositedTotal = 0;
192:         }
```

The misalignment of how the `globalPositions.marginDepositedTotal` should behave within different functions causes various issues within the protocol.

Consider the following scenario to demonstrate one of the negative impacts.

- Alice (LP) deposits 25 ETH and mints 25 UNIT
- LP's `stableCollateralTotal` = 25 ETH
- Bob (Trader) deposits 10 ETH as a margin and opens a long position with the size of 100 ETH (10x leverage)
- `globalPositions.marginDepositedTotal` = 10 ETH (Consists of margin of Bob position)
- Total ETH within the protocol = 35 ETH (25 + 10)

Assume the accrued funding fee is 20 ETH, and the long traders must pay the LP (short). Note that these values are intentionally chosen or inflated to demonstrate the accounting/math error in the implementation, and they do not affect the validity of this issue.

Whenever any of the transactions (e.g. announce or execute order, change protocol's parameters) within the protocol is executed, the following `FlatcoinVault.settleFundingFees` function will be executed to settle the global funding fees. In addition, the `FlatcoinVault.settleFundingFees` function is also permissionless and can be executed by anyone at any point in time.

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L224

```
File: FlatcoinVault.sol
224:     function settleFundingFees() public returns (int256 _fundingFees) {
225:         (int256 fundingChangeSinceRecomputed, int256 unrecordedFunding) =
 ↪  _getUnrecordedFunding();
226:
227:         // Record the funding rate change and update the cumulative funding
 ↪  rate.
228:         cumulativeFundingRate =
 ↪  PerpMath._nextFundingEntry(unrecordedFunding, cumulativeFundingRate);
229:
230:         // Update the latest funding rate and the latest funding
 ↪  recomputation timestamp.
231:         lastRecomputedFundingRate += fundingChangeSinceRecomputed;
232:         lastRecomputedFundingTimestamp = (block.timestamp).toUint64();
233:
234:         // Calculate the funding fees accrued to the longs.
235:         // This will be used to adjust the global margin and collateral
 ↪  amounts.
236:         _fundingFees =
 ↪  PerpMath._accruedFundingTotalByLongs(_globalPositions, unrecordedFunding);
237:
```

```
238:         // In the worst case scenario that the last position which remained
↪  open is underwater,
239:         // We set the margin deposited total to negative. Once the
↪  underwater position is liquidated,
240:         // then the funding fees will be reverted and the total will be
↪  positive again.
241:         _globalPositions.marginDepositedTotal =
↪  _globalPositions.marginDepositedTotal + _fundingFees;
242:
243:         _updateStableCollateralTotal(-_fundingFees);
244:     }
```

Assume that the `FlatcoinVault.settleFundingFees` function is executed at this point. Line 241 above will be evaluated as follows. The `globalPositions.marginDepositedTotal` will be updated to -10 ETH.

```
_globalPositions.marginDepositedTotal = _globalPositions.marginDepositedTotal +
↪  _fundingFees;
_globalPositions.marginDepositedTotal = 10 ETH + (-20 ETH) = -10 ETH
```

In the comments in Lines 238-240 above within the `FlatcoinVault.settleFundingFees` function, it expects the liquidation process to re-adjust back the `globalPositions.marginDepositedTotal` to positive or the correct value later.

Assume that a liquidation is executed. The `vault.updateGlobalPositionData` function at Line 150 within the `LiquidationModule.liquidate` function below will be executed, and the value of `marginDelta` parameter will be as follows:

```
marginDelta = -(int256(position.marginDeposited) +
↪  positionSummary.accruedFunding)
marginDelta = -(10 ETH + (-20 ETH))
marginDelta = -(-10 ETH)
marginDelta = +10 ETH
```

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/LiquidationModule.sol#L150

```
File: LiquidationModule.sol
074:     function liquidate(
..SNIP..
149:         // Update the global leverage position data.
150:         vault.updateGlobalPositionData({
151:             price: position.averagePrice,
152:             marginDelta: -(int256(position.marginDeposited) +
↪  positionSummary.accruedFunding),
```

```
153:                 additionalSizeDelta: -int256(position.additionalSize) // Since
↪  position is being closed, additionalSizeDelta should be negative.
154:             });
```

Within the `updateGlobalPositionData` function below, Line 183 will evaluate as follows:

```
newMarginDepositedTotal = _globalPositions.marginDepositedTotal + _marginDelta;
newMarginDepositedTotal = -10 + (+10) = 0
```

This is aligned with the earlier code's comments (`Once the underwater position is liquidated, then the funding fees will be reverted and the total will be positive again.`) as the `globalPositions.marginDepositedTotal` indeed is restored back to the positive and correct value after the re-adjustment.

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L175

```
File: FlatcoinVault.sol
175:     function updateGlobalPositionData(
176:         uint256 _price,
177:         int256 _marginDelta,
178:         int256 _additionalSizeDelta
179:     ) external onlyAuthorizedModule {
180:         // Note that technically, even the funding fees should be accounted
↪  for when computing the margin deposited total.
181:         // However, since the funding fees are settled at the same time as
↪  the global position data is updated,
182:         // we can ignore the funding fees here.
183:         int256 newMarginDepositedTotal =
↪  _globalPositions.marginDepositedTotal + _marginDelta;
```

Unfortunately, in reality, the execution of the `settleFundingFees` function is not always followed by the execution of the liquidation logic. Thus, the logic code expect the liquidation process executed later to always re-adjust the `globalPositions.marginDepositedTotal` value back to the intended value.

Let's assume another scenario. Assume that after the `FlatcoinVault.settleFundingFees` function is executed, someone triggers an action (e.g., open, adjust, close order) that executes the `updateGlobalPositionData` function instead of the liquidation. Assume that the `_marginDelta` is insignificant in the following:

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L175

```
File: FlatcoinVault.sol
175:     function updateGlobalPositionData(
176:         uint256 _price,
177:         int256 _marginDelta,
178:         int256 _additionalSizeDelta
179:     ) external onlyAuthorizedModule {
180:         // Note that technically, even the funding fees should be accounted
↪  for when computing the margin deposited total.
181:         // However, since the funding fees are settled at the same time as
↪  the global position data is updated,
182:         // we can ignore the funding fees here.
183:         int256 newMarginDepositedTotal =
↪  _globalPositions.marginDepositedTotal + _marginDelta;
..SNIP..
190:         if (newMarginDepositedTotal < 0) {
191:             newMarginDepositedTotal = 0;
192:         }
```

When Line 190 above is executed, since `globalPositions.marginDepositedTotal` is negative (-10 ETH), it will be reset to zero. This is an issue because the code earlier assumes that the liquidation process will help to re-adjust the negative value. However, over here, the state has been wiped out.

When the liquidation is executed subsequently, the `marginDelta` will be `+10 ETH`, similar to the earlier scenario.

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/LiquidationModule.sol#L150

```
File: LiquidationModule.sol
074:     function liquidate(
..SNIP..
149:         // Update the global leverage position data.
150:         vault.updateGlobalPositionData({
151:             price: position.averagePrice,
152:             marginDelta: -(int256(position.marginDeposited) +
↪  positionSummary.accruedFunding),
153:             additionalSizeDelta: -int256(position.additionalSize) // Since
↪  position is being closed, additionalSizeDelta should be negative.
154:         });
```

Within the `updateGlobalPositionData` function, the `globalPositions.marginDepositedTotal` will be set to as follows:

```
newMarginDepositedTotal = _globalPositions.marginDepositedTotal + _marginDelta
```


```
newMarginDepositedTotal = 0 + (+10 ETH)
newMarginDepositedTotal = 10 ETH
```

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L175

```
File: FlatcoinVault.sol
175:     function updateGlobalPositionData(
176:         uint256 _price,
177:         int256 _marginDelta,
178:         int256 _additionalSizeDelta
179:     ) external onlyAuthorizedModule {
180:         // Note that technically, even the funding fees should be accounted
 ↪  for when computing the margin deposited total.
181:         // However, since the funding fees are settled at the same time as
 ↪  the global position data is updated,
182:         // we can ignore the funding fees here.
183:         int256 newMarginDepositedTotal =
 ↪  _globalPositions.marginDepositedTotal + _marginDelta;
```

At the end of the liquidation, the long trader side, which had lost all its margin earlier, regained 10 ETH, which is incorrect and shows an error in the protocol's accounting.

The `globalPositions.marginDepositedTotal` is also inflated and not backed by any assets. The system will wrongly assume that there are `globalPositions.marginDepositedTotal`, while in reality, there is none. This results in the system continuing to transfer collateral assets to the LP when settling funding fees when there are none. The LP will receive collateral credit backed by no assets, leading to a loss of assets for them.

Impact

The `marginDepositedTotal` state is the key variable in the protocol's account system, and its value will be incorrect, leading to the protocol being broken. It will also lead to a loss of assets for the LP.

Code Snippet

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L224

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/FlatcoinVault.sol#L175

Tool used

Manual Review

Recommendation

Ensure that the behavior of `globalPositions.marginDepositedTotal` (whether it can go negative) is consistent throughout the codebase to prevent any error from occurring.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

itsermin

Yes, it looks like when `marginDepositedTotal` is negative, opening a new leverage position can break an invariant because of setting it to 0 on `updateGlobalPositionData`.

By removing this line (not zeroing a negative `marginDepositedTotal`), the system is able to handle the scenario.

santipu03

Escalate

I believe this issue to be medium severity due to the low probability of happening.

For this bug to be triggered it needs the value of `globalPositions.marginDepositedTotal` to be negative. These scenarios can cause it:

1. The total debt of the positions on the market becomes bigger than the total deposited margin.
2. The funding fee becomes bigger than the total deposited margin.
3. A combination of both scenarios above.

Scenario 1 is highly unlikely to happen because insolvent positions would have been liquidated before their debt became bigger than the total deposited margin on the market. Moreover, in a volatile market, the protocol team can adjust the value of `liquidationBufferRatio` to ensure that even sharp price movements don't cause positions to get underwater.

Scenario 2 is also highly unlikely to happen because when the funding fee is so big, it's expected that the market will regulate itself, bringing the skew back to zero and decreasing the funding fees. Even if the funding fee starts to rapidly decrease the margins of the positions, liquidators will step up and liquidate those underwater positions before bad debt is accrued.

Given the low probability of this bug happening, I think it deserves medium severity.

sherlock-admin2

> Escalate
>
> I believe this issue to be medium severity due to the low probability of happening.
>
> For this bug to be triggered it needs the value of `globalPositions.marginDepositedTotal` to be negative. These scenarios can cause it:
>
> 1. The total debt of the positions on the market becomes bigger than the total deposited margin.
> 2. The funding fee becomes bigger than the total deposited margin.
> 3. A combination of both scenarios above.
>
> Scenario 1 is highly unlikely to happen because insolvent positions would have been liquidated before their debt became bigger than the total deposited margin on the market. Moreover, in a volatile market, the protocol team can adjust the value of `liquidationBufferRatio` to ensure that even sharp price movements don't cause positions to get underwater.
>
> Scenario 2 is also highly unlikely to happen because when the funding fee is so big, it's expected that the market will regulate itself, bringing the skew back to zero and decreasing the funding fees. Even if the funding fee starts to rapidly decrease the margins of the positions, liquidators will step up and liquidate those underwater positions before bad debt is accrued.
>
> Given the low probability of this bug happening, I think it deserves medium severity.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

cvetanovv

I agree with the low probability, but when it happens it definitely hits the High category on both points according to the rules:

- Definite loss of funds without (extensive) limitations of external conditions.
- Inflicts serious non-material losses.

santipu03

The README states that the protocol is running liquidators, so it's assumed that unhealthy positions are going to be liquidated on time and bad debt is not going to be created:

> Are there any off-chain mechanisms or off-chain procedures for the protocol (keeper bots, arbitrage bots, etc.)?
>
> - There are keepers for order execution and liquidations. [...]

I'd say that this itself is an extensive limitation for this issue to happen.

WangSecurity

Even though the protocol will be using keepers for order execution and liquidations, it doesn't ensure that every liquidatable position will be liquidated. Despite that fact, the issue will happen in the scenarios listend in the escalation comment here, hence, I believe medium is appropriate here cause it requires specific states, and loss is highly constrained:

> Causes a loss of funds but requires certain external conditions or specific states, or a loss is highly constrained.

Planning to accept the escalation and downgrade the issue to Medium.

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- santipu03: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/flatcoin-v1/pull/344

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: Attacker can steal LPs funds by using different oracle prices in the same transaction

Source: https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest-judging/issues/27

The protocol has acknowledged this issue.

Found by

bareli, santipu_, xiaoming90

Summary

In the previous audit, there was an issue that pointed out that the PYTH oracle can return different prices within the same transaction. The fix implemented was to prevent the attack path described there, but a malicious LP can steal funds using a similar attack path.

Vulnerability Detail

The attack path is as follows:

1. The attacker has some liquidity deposited as an LP with address A.
2. The attacker announces the withdrawal of all liquidity with address A.
3. In the same txn, the attacker announces the deposit of liquidity (same amount as the withdrawal) with address B.
4. After the minimum execution time has elapsed, retrieve two prices from the Pyth oracle where the second price is lower than the first one.
5. Execute the withdrawal of address A sending the higher price.
6. Execute the deposit of address B sending the lower price.

As a result, the attacker will have the same deposited liquidity in the protocol, plus a profit from the price deviation.

Even though the withdrawal may reduce the profits of this attack, it doesn't completely prevent it. Moreover, if the attacker is the only LP on the market, there's not going to be a withdrawal fee, increasing the profits of such an attack at the expense of the traders.

The requirements of this attack are the same as the ones described in the original issue, hence the medium severity.

Sidenote: Now that I'm checking better the fixes on the previous audit, I think that the second attack path described by the original issue is still not fixed:

> Another possible strategy would pass through the following steps:
>
> - Create a leverage position.
> - Announce another leverage position with the same size.
> - In the same block, announce a limit close order.
> - After the minimum execution time has elapsed, retrieve two prices from the Pyth oracle where the second price is lower than the first one.
> - Execute the limit close order sending the first price.
> - Execute the open order sending the second price.
>
> The result in this case is having a position with the same size as the original one, but having either lowered the position.lastPrice or getting a profit from the original position, depending on how the price has moved since the original position was opened.

I don't have access to the exact pull request of the fix but I'm pretty sure the attack path described above is still feasible.

Impact

Different oracle prices can be fetched in the same transaction, which can be used to create arbitrage opportunities that can make a profit with no risk at the expense of users on the other side of the trade.

Code Snippet

https://github.com/sherlock-audit/2024-03-flat-money-fix-review-contest/blob/main/flatcoin-v1/src/OracleModule.sol#L63

Tool used

Manual Review

Recommendation

Given the time constraints of this contest, I haven't been able to think about a quick fix for this issue. Even if the protocol team doesn't allow to update the price feeds within the same block, an attacker can still update them by directly calling the PYTH contract.

Discussion

sherlock-admin4

1 comment(s) were left on this issue during the judging contest.

santipu_ commented:

> Medium

rashtrakoff

The original scenario is not possible imo because the limit order's execution time is changed when a delayed order adjusts a position. However, I think other variations like the one you described is still possible.

I think to only solution would be to use the same price in a block for all orders. This would still allow for multi-block arbitrage attacks but that will be incredibly difficult to perform and probably will result in losses mostly.

itsermin

I think this scenario is part of a range of oracle frontrunning issues. It doesn't neccesarily require 2 prices in a single block to profit.

In a simpler example issue, the attacker:

- Announces a leverage open
- Fetches 2 Pyth prices
- Executes open on the lower price
- Announces a close
- Fetches 2 Pyth prices
- Executes close on the higher price

Just like the original example, this example is profitable over time if it wasn't for the trading fee that the attacker has to pay on each order. Whether the original example steps 5 and 6 are done in a single block or consecutive blocks is also not critical either.

The trading fee mitigates any of these types of oracle frontrunning issues from being profitable.

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.