SHERLOCK

# SHERLOCK SECURITY REVIEW FOR

SHERLOCK

| | |
|---|---|
| Prepared for: | Float Capital |
| Prepared by: | Sherlock |
| Lead Security Expert: | WATCHPUG |
| Dates Audited: | November 2 - November 9, 2022 |
| Prepared on: | November 22, 2022 |

# Introduction

Float is tokenized long/short engine with a multi-pool architecture. Arctic ensures no liquidations and predictable tokenized leveraged exposure.

## Scope

The following files are in scope

```
./contracts/oracles/OracleManager.sol
./contracts/YieldManagers/MarketLiquidityManagerSimple.sol
./contracts/market/template/MarketExtended.sol
./contracts/market/template/MarketCore.sol
./contracts/PoolToken/PoolToken.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

| Medium | High |
|---|---|
| 2 | 1 |

## Issues not fixed or acknowledged

| Medium | High |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

WATCHPUG
ctf_sec
pashov
0x52

## Issue H-1: An update gap in Chainlink's feed can malfunction the whole market

Source: https://github.com/sherlock-audit/2022-11-float-capital-judging/issues/42

### Found by

WATCHPUG

### Summary

The `roundId` that is used for settling the price change and pushing the `latestExecutedEpochIndex` forward is strictly limited to be in a precise period of time. When there is no such `roundId`, the system will freeze and lock everyone out.

### Vulnerability Detail

The check at L127 makes it impossible to use a roundId that was created at a later time than `relevantEpochStartTimestampWithMEWT+EPOCH_LENGTH`.

However, when the `EPOCH_LENGTH` is larger than the Chainlink feed's heartbeat length, or Chainlink failed to post a feed within the expected heartbeat for whatever reason, then it would be impossible to find a suitable roundId (as it does not exist) to push the epoch forward due to the rather strict limitation for the roundId.

### Impact

As a result, the whole system will malfunction and no one can enter or exit the market.

### Code Snippet

https://github.com/sherlock-audit/2022-11-float-capital/blob/main/contracts/market/template/MarketCore.sol#L188-L195

### Tool used

Manual Review

### Recommendation

Consider allowing the `roundId` not to falls into the epoch, and use the previous roundId's price when that's the case:

```
    for (uint32 i = 0; i < lengthOfEpochsToExecute; i++) {
      // Get correct data
    (, int256 currentOraclePrice, uint256 currentOracleUpdateTimestamp, , ) =
↪ chainlinkOracle.getRoundData(oracleRoundIdsToExecute[i]);

    // Get Previous round data to validate correctness.
-    (, , uint256 previousOracleUpdateTimestamp, , ) =
↪ chainlinkOracle.getRoundData(oracleRoundIdsToExecute[i] - 1);
+    (, int256 previousOraclePrice, uint256 previousOracleUpdateTimestamp, , )
↪ = chainlinkOracle.getRoundData(oracleRoundIdsToExecute[i] - 1);

    // Check if there was a 'phase change' AND the `_currentOraclePrice` is
↪ zero.
    if ((oracleRoundIdsToExecute[i] >> 64) > (latestExecutedOracleRoundId >>
↪ 64) && previousOracleUpdateTimestamp == 0) {
      // NOTE: if the phase changes, then we want to correct the phase of the
↪ update.
      //       There is no guarantee that the phaseID won't increase multiple
↪ times in a short period of time (hence the while loop).
      //       But chainlink does promise that it will be sequential.
      // View how phase changes happen here:
↪ https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.7/dev/Aggregato
      while (previousOracleUpdateTimestamp == 0) {
        // NOTE: re-using this variable to keep gas costs low for this edge
↪ case.
        latestExecutedOracleRoundId = (((latestExecutedOracleRoundId >> 64) +
↪ 1) << 64) | uint64(oracleRoundIdsToExecute[i] - 1);

        (, , previousOracleUpdateTimestamp, , ) =
↪ chainlinkOracle.getRoundData(latestExecutedOracleRoundId);
      }
    }

    // This checks the price given is valid and falls within the correct
↪ window.
    // see https://app.excalidraw.com/l/2big5WYTyfh/4PhAp1a28s1
    if (
      previousOracleUpdateTimestamp >= relevantEpochStartTimestampWithMEWT ||
      currentOracleUpdateTimestamp < relevantEpochStartTimestampWithMEWT
-      currentOracleUpdateTimestamp >= relevantEpochStartTimestampWithMEWT +
↪ EPOCH_LENGTH
    ) revert InvalidOracleExecutionRoundId({oracleRoundId:
↪ oracleRoundIdsToExecute[i]});

+    // If the new roundId does not falls into the epoch, use the prev roundId
↪ then
+    if (currentOracleUpdateTimestamp >= relevantEpochStartTimestampWithMEWT +
↪ EPOCH_LENGTH) {
+      currentOraclePrice = previousOraclePrice;
+    }
```

```
      if (currentOraclePrice <= 0) revert InvalidOraclePrice({oraclePrice:
↪   currentOraclePrice});

      missedEpochPriceUpdates[i] = currentOraclePrice;

      relevantEpochStartTimestampWithMEWT += EPOCH_LENGTH;
    }
```

## Discussion

**JasoonS**

Thanks - we had a long internal debate discussion about this.

We decided that it is best to make the parameters (such as epoch length) long enough such that this is extremely unlikely to happen.

We have done some extensive latency and heartbeat analysis on chainlink oracles - as well as had an in-details discussion about how gas price spikes can cause delays to prices being pushed on chain (a side note - this is why a large mewt/minimumExecutionWatingTi is required - otherwise a gas price spike/griefing attack would be more feasible). I'll link some of that to this issue in a bit if that is interesting to you.

Anyway - getting back to this issue - we believe it is better to leave the market paused if such an anomaly happens and give us time to analyse what happened. It is a sort of risk protection mechanism. Either we upgrade market for a fix (which will be under timelock), or we deprecate the market.

I think our users will appreciate our prudence.

One thing to consider is that withdrawals also won't be processed in this edge case (maybe a good think?). I'll have another chat with the team on that.

Agree that your solution is pretty benign too since there will just be no price change.

**moose-code**

@WooSungD would be useful if you could post that graph of chainlink prices on the analysis we did.

**moose-code**

For more context, a few weeks ago we had detailed disscussion with the chainlink team, as you can't even rely on the hearbeat with certainty.

E.g. the heartbeat of 27sec on polygon still showed outliers where we waited for up to 180 seconds in some cases for a new price because of big gas spikes. This is why we conducted the analysis so carefully, we want to make sure that we don't miss a chainlink price.

However if we do miss a price, the auto deprecation means the system fails very gracefully, the markets are paused and everyone can simply withdraw after a cooldown period.

**WooSungD**

Here are some graphs showing the distribution of heartbeat (in seconds) for ETH-USD price feed on Chainlink Polygon.

The outliers for the heartbeat mean that our MEWT needs to be longer (longer than max outlier necessarily) to prevent front-running.

The causes of outliers ito heartbeat were network congestion and gas spikes, according to the Chainlink team

**moose-code**

After chatting with the chainlink team more on this, the one potential attack vector (that seems unrealistic) that I can point out is spamming the polygon chain to the point where it delays the chainlink price update from being mined until the point where no valid price exists.

This would be extremely expensive and simply cause the market to deprecate (no financial gain).

**Evert0x**

We still think this is a high severity issue as it can make the protocol malfunction

# Issue M-1: Unsafe type casting of `poolValue` can malfunction the whole market

Source: https://github.com/sherlock-audit/2022-11-float-capital-judging/issues/45

## Found by

WATCHPUG

## Summary

When `poolValue` is a negative number due to loss in `valueChange` and `funding`, the unsafe type casting from `int256` to `uint256` will result in a huge number close to `2**255` which will revert `_rebalancePoolsAndExecuteBatchedActions()` due to overflow when multiplied by 1e18 at L163.

## Vulnerability Detail

If the funding rate is 100% per year and the `EPOCH_LENGTH` is 4 days, the funding fee for each epoch can be as much as ~1% on the effectiveValue.

Plus, the loss from `valueChange` is capped at 99%, but combining both can still result in a negative `poolValue` at L146.

At L163 `uint256price=uint256(poolValue).div(tokenSupply);` the type casting from `int256` to `uint256` will result in a huge number close to `2**255`.

`MathUintFloat.div()` will overflow when a number as large as `2**255` is multiplied by 1e18.

## Impact

`_rebalancePoolsAndExecuteBatchedActions` will revert and cause the malfunction of the whole market.

## Code Snippet

https://github.com/sherlock-audit/2022-11-float-capital/blob/main/contracts/market/template/MarketCore.sol#L118-L185

## Tool used

Manual Review

## Recommendation

Consider adding a new function to properly handle the bankruptcy of a specific pool.

## Discussion

**JasoonS**

We seed the pools initially with sufficient un-extractable capital such that this shouldn't be an issue (it should never get close to 0 - even after millions of years and trillions of transactions that may have rounding down and all users withdrawing their funds).

We could create a safe cast function to check - but we made `poolValue` an int256 so that it is easier to operate on with other signed integers - not because it is ever possible for it to be negative. So it would be redundant in this case.

**moose-code**

@JasoonS Want to relook at this. @WooSungD @Stentonian maybe you also have thoughts.

I believe watchpug is explaining something different.

They are saying that poolValue can be negative, as a 99% capped loss of poolValue, in conjunction with a 1% funding fee (imagine the side is very overbalanced), will result in the pool value losing more than 100% in total.

A safe guard would be to check that with BOTH funding and value change, 99% is the maximum a pool can lose in any single iteration.

Given system parameterizations, where epoch length will never be that long and funding rate should never be that high, its unlikely this would be an issue in practice, but likely still worth making a change for.

Let me know if anyone has thoughts

**JasoonS**

Yes, you're right, went through these too fast.

We've discussed this internally a few times. This point should've made it into the readme.

We could add checks to the epoch length on construction to ensure were safe

## Issue M-2: Protocol won't work with `USDC` even though it is a token specifically mentioned in the docs

Source: https://github.com/sherlock-audit/2022-11-float-capital-judging/issues/21

### Found by

pashov, ctf_sec, 0x52

### Summary

The protocol has requirements for values (for example 1e18) that would be too big if used with a 6 decimals token like `USDC` - `USDC` is mentioned as a token that will be used in the docs

### Vulnerability Detail

For the mint functionality, a user has to transfer at least 1e18 tokens so that he can mint pool tokens - `if(amount<1e18)revertInvalidActionAmount(amount);`. If the `paymentToken` used was `USDC` (as pointed out in docs), this would mean he would have to contribute at least 1e12 USDC tokens (more than a billion) which would be pretty much impossible to do. There is also another such check in `MarketExtended::addPoolToExistingMarket` with `require(initialActualLiquidityForNewPool>=1e12,"Insufficientmarketseed");` - both need huge amounts when using a low decimals token like USDC that has 6 decimals.

### Impact

The protocol just wouldn't work at all in its current state when using a lower decimals token. Since such a token was mentioned in the docs I set this as a High severity issue.

### Code Snippet

https://github.com/sherlock-audit/2022-11-float-capital/blob/main/contracts/market/template/MarketExtended.sol#L125 https://github.com/sherlock-audit/2022-11-float-capital/blob/main/contracts/market/template/MarketCore.sol#L265

### Tool used

Manual Review

## Recommendation

Drastically lower the `require` checks so they can work with tokens with a low decimals count like `USDC`

## Discussion

**JasoonS**

I feel really silly that I didn't think of that when I wrote the readme - we have spoken about it came up many times in the alpha version audit that we did last year.  We have no intention of using USDC anytime soon. We have been using DAI exclusively. My mistake - I thought why not just have the option for insurance sake and mention USDC (since it is the only other token remotely likely that we might use).

This most certainly isn't `high` - it is in the constructor that we'd immediately notice that (of course as I mentioned we have been aware of this for a long time). The rest of the mechanism works with USDC - just those minimums will need to be adjusted.

So, "*Bug in the readme?*" I'd say this isn't a vulnerability at all!

We could fetch the `decimals` from the payment token on initialization, but honestly don't think the extra complexity is justified in our situation.

**Evert0x**

As USDC was explicitly mentioned by the protocol we would like to reward this finding.