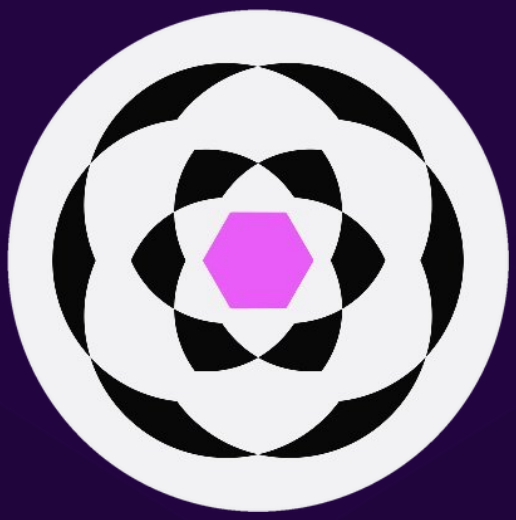
SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for:	Perennial
Prepared by:	Sherlock
Lead Security Expert:	panprog
Dates Audited:	September 18 - September 23, 2023
Prepared on:	October 2, 2023

Introduction

Perennial is built from first principles as a powerful DeFi primitive that scales to meet the needs of traders, LPs, and developers.

Scope

Repository: equilibria-xyz/root

Branch: v2

Commit: d531cf7f0c1d417e9987b706092e177021a89e5d

Repository: equilibria-xyz/perennial-v2

Branch: main

Commit: 3e7c37d42a19f2f1c262d4059bbcafe7d37c5796

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
7	0

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

panprog

bin2chen

Issue M-1: During oracle provider switch, if previous provider feed stops working completely, oracle and market will be stuck with user funds locked in the contract

Source: https://github.com/sherlock-audit/2023-09-perennial-judging/issues/10

Found by

panprog

The issue 46 of the main contest after the fix still stands with a more severe condition as described by WatchPug in fix review:

> If we assume it's possible for the previous Python feed to experience a more severe issue: instead of not having an eligible price for the requested oracleVersion, the feed completely stopped working after the requested time, making it impossible to find ANY valid price at a later time than the last requested time, this issue would still exist.

Sponsor response still implies that the previous provider feed **is available**, as they say non-requested version could be posted, but if this feed is no longer available, it will be impossible to commit unrequested, because there will be no pyth price and signature to commit.

> if the previous oracle's underlying off-chain feed goes down permanently, once the grace period has passed, a non-requested version could be posted to the previous oracle, moving its latest() forward to that point, allowing the switchover to complete.

Vulnerability Detail

When the oracle provider is updated (switched to a new provider), the latest status (price) returned by the oracle will come from the previous provider until the last request is commited for it, only then the price feed from the new provider will be used. However, it can happen that pyth price feed stops working completely before (or just after) the oracle is updated to a new provider. This means that valid price with signature for **any timestamp after the last request** is not available. In this case, the oracle price will be stuck, because it will ignore new provider, but the previous provider can never finalize (commit a fresh price). As such, the oracle price will get stuck and will never update, breaking the whole protocol with user funds stuck in the protocol.

Impact

Switching oracle provider can make the oracle stuck and stop updating new prices. This will mean the market will become stale and will revert on all requests from

user, disallowing to withdraw funds, bricking the contract entirely.

Code Snippet

`Oracle._latestStale` will always return false due to this line (since `latest().timestamp` can never advance without a price feed): https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/Oracle.sol#L128

Tool used

Manual Review

Recommendation

Consider ignoring line 128 in `Oracle._latestStale` if a certain timeout has passed after the switch (`block.timestamp - oracles[global.latest].timestamp > SWITCH_TIMEOUT`). This will allow the switch to proceed after some timeout even if previous provider remains uncommited.

Discussion

sherlock-admin

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

polarzero commented:

> Medium. The issue is also preoccupying. It does also require a few unlikely conditions, yet it could incur a significant loss of funds for the users.

Issue M-2: commitRequested() front-run malicious invalid oralce

Source: https://github.com/sherlock-audit/2023-09-perennial-judging/issues/27

Found by

bin2chen Both `commitRequested()` and `commit()` can modify `lastCommittedPublishTime`, and both check that they cannot `pythPrice.publishTime<=lastCommittedPublishTime`. This allows a malicious user to front-run `commitRequested()` to execute `commit()`, causing `commitRequested()` to revert, invalid oralce

Vulnerability Detail

Execution of the `commitRequested()` method restricts the `lastCommittedPublishTime` from going backward.

```
    function commitRequested(uint256 versionIndex, bytes calldata updateData)
        public
        payable
        keep(KEEPER_REWARD_PREMIUM, KEEPER_BUFFER, updateData, "")
    {
...

@>      if (pythPrice.publishTime <= lastCommittedPublishTime) revert
↪   PythOracleNonIncreasingPublishTimes();
@>        lastCommittedPublishTime = pythPrice.publishTime;
...
```

`commit()` has a similar limitation and can set `lastCommittedPublishTime`.

```
    function commit(uint256 versionIndex, uint256 oracleVersion, bytes calldata
↪   updateData) external payable {
        if (
            versionList.length > versionIndex &&                // must be a
↪   requested version
            versionIndex >= nextVersionIndexToCommit &&         // must be the
↪   next (or later) requested version
@>          oracleVersion == versionList[versionIndex]          // must be the
↪   corresponding timestamp
        ) {
            commitRequested(versionIndex, updateData);
            return;
        }
...
```

```
@>      if (pythPrice.publishTime <= lastCommittedPublishTime) revert
↪  PythOracleNonIncreasingPublishTimes();
@>      lastCommittedPublishTime = pythPrice.publishTime;
....
```

This leads to a situation where anyone can front-run `commitRequested()` and use his `updateData` to execute `commit()`. In order to satisfy the `commit()` constraint, we need to pass a `commit()` parameter set as follows

1. versionIndex= nextVersionIndexToCommit
2. oracleVersion = versionList[versionIndex] - 1 and oralceVersion > _latestVersion
3. pythPrice.publishTime >= versionList[versionIndex] - 1 + MIN_VALID_TIME_AFTER_VERSION

This way `lastCommittedPublishTime` will be modified, causing `commitRequested()` to execute with `revert PythOracleNonIncreasingPublishTimes`

Example: Given: nextVersionIndexToCommit = 10 versionList[10] = 200 _latestVersion = 100

when:

1. keeper exexute commitRequested(versionIndex = 10 , VAA{ publishTime = 205})
2. front-run execute ‘commit(versionIndex = 10 , oracleVersion = 200-1 , VAA{ publishTime = 205})
    - versionIndex= nextVersionIndexToCommit pass
    - oracleVersion = versionList[versionIndex] - 1 and oralceVersion > _latestVersion (pass)
    - pythPrice.publishTime >= versionList[versionIndex] - 1 + MIN_VALID_TIME_AFTER_VERSION (pass)

By the time the `keeper` submits the next VVA, the price may have passed its expiration date

Impact

If the user can control the oralce invalidation, it can lead to many problems e.g. invalidating `oracle` to one's own detriment, not having to take losses Maliciously destroying other people's profits, etc.

Code Snippet

https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L174

https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L129

Tool used

Manual Review

Recommendation

check `pythPrice` whether valid for `nextVersionIndexToCommit`

```
    function commit(uint256 versionIndex, uint256 oracleVersion, bytes calldata
↪   updateData) external payable {
        // Must be before the next requested version to commit, if it exists
        // Otherwise, try to commit it as the next request version to commit
        if (
            versionList.length > versionIndex &&                // must be a
↪   requested version
            versionIndex >= nextVersionIndexToCommit &&         // must be the
↪   next (or later) requested version
            oracleVersion == versionList[versionIndex]          // must be the
↪   corresponding timestamp
        ) {
            commitRequested(versionIndex, updateData);
            return;
        }

        PythStructs.Price memory pythPrice = _validateAndGetPrice(oracleVersion,
↪   updateData);

        // Price must be more recent than that of the most recently committed
↪   version
        if (pythPrice.publishTime <= lastCommittedPublishTime) revert
↪   PythOracleNonIncreasingPublishTimes();
        lastCommittedPublishTime = pythPrice.publishTime;

        // Oracle version must be more recent than that of the most recently
↪   committed version
        uint256 minVersion = _latestVersion;
        uint256 maxVersion = versionList.length > versionIndex ?
↪   versionList[versionIndex] : current();
```

```
        if (versionIndex < nextVersionIndexToCommit) revert
↪   PythOracleVersionIndexTooLowError();
        if (versionIndex > nextVersionIndexToCommit && block.timestamp <=
↪   versionList[versionIndex - 1] + GRACE_PERIOD)
            revert PythOracleGracePeriodHasNotExpiredError();
        if (oracleVersion <= minVersion || oracleVersion >= maxVersion) revert
↪   PythOracleVersionOutsideRangeError();
+       if (nextVersionIndexToCommit < versionList.length) {
+           if (
+               pythPrice.publishTime >= versionList[nextVersionIndexToCommit] +
↪   MIN_VALID_TIME_AFTER_VERSION &&
+               pythPrice.publishTime <= versionList[nextVersionIndexToCommit] +
↪   MAX_VALID_TIME_AFTER_VERSION
+           ) revert PythOracleUpdateValidForPreviousVersionError();
+       }


        _recordPrice(oracleVersion, pythPrice);
        nextVersionIndexToCommit = versionIndex;
        _latestVersion = oracleVersion;
    }
```

Discussion

sherlock-admin

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

panprog commented:

> borderline low/medium. The issue is valid and can force keepers to re-submit if they're frontrun. It's still always possible to submit a price with publishTime which is at MAX_VALID_TIME_AFTER_VERSION away from version time, but this still interfere oracle keepers process and increases chances of invalid version. Definitely not high, because it doesn't break things, just forces to re-submit transactions and keepers can also front-run each other, so reverted keep transactions are not something possible only due to this issue. Probably a better fix is to commitRequested instead of just commit if publishTime is between MIN and MAX valid time.

n33k commented:

> invalid, expected behavior for commitRequested to revert because commit alreay provided the oracleVersion

0xyPhilic commented:

> invalid because there is no proof of funds loss

polarzero commented:

> Medium. Not sure what the incentive would be for an attacker to do this, and the impact it would have, but I'd rather have it downgraded than ignored.

Issue M-3: `Vault.update(anyUser,0,0,0)` can be called for free to increase `checkpoint.count` and pay smaller keeper fee than necessary

Source: https://github.com/sherlock-audit/2023-09-perennial-judging/issues/29

Found by

panprog

Vault re-balances its deposits and redemptions once per oracle epoch, but since multiple users can deposit/redeem, rebalance keeper fee is shared equally between all users depositing and redeeming in the same epoch. For this reason, `checkpoint.count` counts the number of users who will split the keeper fee (each user pays `keeper fee / checkpoint.count`)

The problem is that there are currently 2 types of users who increase `checkpoint.count` while they do not pay any fee:

1. Any user calling `Vault.update(user, 0, 0, 0)` isn't charged any fee but increases `checkpoint.count`
2. Any user claiming assets is charged full `settlementFee` from the amount he claims, but is not charged any other (shared) fees after that, but still increases `checkpoint.count`

The 1st point is more severe, because it allows to intentionally reduce the keeper fee paid by calling `Vault.update(0,0,0)` from different random accounts (which costs only gas fees and nothing more). Each such call increases `checkpoint.count` and thus reduces the keeper fees paid by the attacker.

Attack scenario:

1. User deposits or withdraws from his normal account. `checkpoint.count = 1` for the epoch. Normally the user will pay the sum of the `settlementFee` of all the vault markets.
2. User uses any 3 addresses without any funds other than ETH for gas to call `Vault.update(address1/2/3, 0,0,0)`. Each of these calls costs only gas to the user, but increases `checkpoint.count` to the value of 4.
3. Once the epoch settles, user will only pay `settlementFee / 4` for his deposit/withdrawal, but the vault will still pay the Markets full `settlementFee` at the expense of the other vault users.

Vulnerability Detail

`Vault._update(user, 0, 0, 0)` will pass all invariants checks:

```
// invariant
// @audit operator - pass
if (msg.sender != account &&
↪  !IVaultFactory(address(factory())).operators(account, msg.sender))
    revert VaultNotOperatorError();
// @audit 0,0,0 is single-sided - pass
if (!depositAssets.add(redeemShares).add(claimAssets).eq(depositAssets.max(redee
↪  mShares).max(claimAssets)))
    revert VaultNotSingleSidedError();
// @audit depositAssets == 0 - pass
if (depositAssets.gt(_maxDeposit(context)))
    revert VaultDepositLimitExceededError();
// @audit redeemShares == 0 - pass
if (redeemShares.gt(_maxRedeem(context)))
    revert VaultRedemptionLimitExceededError();
// @audit depositAssets == 0 - pass
if (!depositAssets.isZero() && depositAssets.lt(context.settlementFee))
    revert VaultInsufficientMinimumError();
// @audit redeemShares == 0 - pass
if (!redeemShares.isZero() && context.latestCheckpoint.toAssets(redeemShares,
↪  context.settlementFee).isZero())
    revert VaultInsufficientMinimumError();
// @audit since this will be called by **different** users in the same epoch,
↪  this will also pass
if (context.local.current != context.local.latest) revert
↪  VaultExistingOrderError();
```

It then calculates amount to claim by calling `_socialize`:

```
// asses socialization and settlement fee
UFixed6 claimAmount = _socialize(context, depositAssets, redeemShares,
↪  claimAssets);
...
function _socialize(
    Context memory context,
    UFixed6 depositAssets,
    UFixed6 redeemShares,
    UFixed6 claimAssets
) private view returns (UFixed6 claimAmount) {
    // @audit global assets must be 0 to make (0,0,0) pass this function
    if (context.global.assets.isZero()) return UFixed6Lib.ZERO;
    UFixed6 totalCollateral =
↪  UFixed6Lib.from(_collateral(context).max(Fixed6Lib.ZERO));
    claimAmount = claimAssets.muldiv(totalCollateral.min(context.global.assets),
↪  context.global.assets);
```

```
    // @audit for (0,0,0) this will revert (underflow)
    if (depositAssets.isZero() && redeemShares.isZero()) claimAmount =
↪   claimAmount.sub(context.settlementFee);
}
```

`_socialize` will immediately return 0 if `context.global.assets == 0`. If `context.global.assets > 0`, then this function will revert in the last line due to underflow (trying to subtract `settlementFee` from 0 `claimAmount`)

This is the condition for this issue to happen: global assets must be 0. Global assets are the amounts redeemed but not yet claimed by users. So this can reasonably happen in the first days of the vault life, when users mostly only deposit, or claim everything they withdraw.

Once this function passes, the following lines increase `checkpoint.count`:

```
// update positions
context.global.update(context.currentId, claimAssets, redeemShares,
↪   depositAssets, redeemShares);
context.local.update(context.currentId, claimAssets, redeemShares,
↪   depositAssets, redeemShares);
context.currentCheckpoint.update(depositAssets, redeemShares);
...
// Checkpoint library:
...
function update(Checkpoint memory self, UFixed6 deposit, UFixed6 redemption)
↪   internal pure {
    (self.deposit, self.redemption) = (self.deposit.add(deposit),
↪   self.redemption.add(redemption));
    self.count++;
}
```

The rest of the function executes normally.

During position settlement, pending user deposits and redeems are reduced by the `keeper fees / checkpoint.count`:

```
// Account library:
...
function processLocal(
    Account memory self,
    uint256 latestId,
    Checkpoint memory checkpoint,
    UFixed6 deposit,
    UFixed6 redemption
) internal pure {
    self.latest = latestId;
    (self.assets, self.shares) = (
```

```
        self.assets.add(checkpoint.toAssetsLocal(redemption)),
        self.shares.add(checkpoint.toSharesLocal(deposit))
    );
    (self.deposit, self.redemption) = (self.deposit.sub(deposit),
↪   self.redemption.sub(redemption));
}
...
// Checkpoint library
// toAssetsLocal / toSharesLocal calls _withoutKeeperLocal to calculate keeper
↪   fees:
...
    function _withoutKeeperLocal(Checkpoint memory self, UFixed6 amount) private
↪   pure returns (UFixed6) {
        UFixed6 keeperPer = self.count == 0 ? UFixed6Lib.ZERO :
↪   self.keeper.div(UFixed6Lib.from(self.count));
        return _withoutKeeper(amount, keeperPer);
    }
```

Also notice that in `processLocal` the only thing which keeper fees influence are deposits and redemptions, but not claims.

Proof of concept

The scenario above is demonstrated in the test, add this to Vault.test.ts:

```
it('inflate checkpoint count', async () => {
    const settlementFee = parse6decimal('10.00')
    const marketParameter = { ...(await market.parameter()) }
    marketParameter.settlementFee = settlementFee
    await market.connect(owner).updateParameter(marketParameter)
    const btcMarketParameter = { ...(await btcMarket.parameter()) }
    btcMarketParameter.settlementFee = settlementFee
    await btcMarket.connect(owner).updateParameter(btcMarketParameter)

    const deposit = parse6decimal('10000')
    await vault.connect(user).update(user.address, deposit, 0, 0)
    await updateOracle()
    await vault.settle(user.address)

    const deposit2 = parse6decimal('10000')
    await vault.connect(user2).update(user2.address, deposit2, 0, 0)

    // inflate checkpoint.count
    await vault.connect(btcUser1).update(btcUser1.address, 0, 0, 0)
    await vault.connect(btcUser2).update(btcUser2.address, 0, 0, 0)

    await updateOracle()
```

```
    await vault.connect(user2).settle(user2.address)

    const checkpoint2 = await vault.checkpoints(3)
    console.log("checkpoint count = " + checkpoint2.count)

    var account = await vault.accounts(user.address);
    var assets = await vault.convertToAssets(account.shares);
    console.log("User shares:" + account.shares + " assets: " + assets);
    var account = await vault.accounts(user2.address);
    var assets = await vault.convertToAssets(account.shares);
    console.log("User2 shares:" + account.shares + " assets: " + assets);
})
```

Console output:

```
checkpoint count = 3
User shares:10000000000 assets: 9990218973
User2 shares:10013140463 assets: 10003346584
```

So the user2 inflates his deposited amounts by paying smaller keeper fee.

If 2 lines which inflate checkpoint count (after corresponding comment) are deleted, then the output is:

```
checkpoint count = 1
User shares:10000000000 assets: 9990218973
User2 shares:9999780702 assets: 9989999890
```

So if not inflated, user2 pays correct amount and has roughly the same assets as user1 after his deposit.

Impact

Malicious vault user can inflate `checkpoint.count` to pay much smaller keeper fee than they should at the expense of the other vault users.

Code Snippet

`Vault.update(0,0,0)` will increase `checkpoint.count` here: https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L300

Tool used

Manual Review

Recommendation

Consider reverting (0,0,0) vault updates, or maybe redirecting to `settle` in this case. Additionally, consider updating checkpoint only if `depositAssets` or `redeemShares` are not zero:

```
if (!depositAssets.isZero() || !redeemShares.isZero())
    context.currentCheckpoint.update(depositAssets, redeemShares);
```

Discussion

sherlock-admin

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

polarzero commented:

> Medium. Perfectly explained and demonstrated in the report.

kbrizzle

Fixed in: https://github.com/equilibria-xyz/perennial-v2/pull/111.

panprog

1. `Vault(user,0,0,0)` increasing `checkpoint.count` (medium severity) - fixed
2. Any user who is claiming still increases `checkpoint.count`, although he pays for the claim fully and doesn't participate in paying the keeper fee for deposit/redeem, thus he shouldn't increase `checkpoint.count` (low severity) - not fixed

kbrizzle

Since (2) is overcharging the fee instead of undercharging (claim pays entire settlement fee instead of splitting the resulting fee with others in version), we're going to leave this as-is. We'll make a note of this in case we improve the claiming flow in the future.

jacksanford1

Based on @kbrizzle's comment, Sherlock will consider the issue brought up in #2 in the comment above by panprog as acknowledged.

Issue M-4: MultiInvoker liquidation action will revert most of the time due to incorrect closable amount initialization

Source: https://github.com/sherlock-audit/2023-09-perennial-judging/issues/44

Found by

panprog

The fix to issue 49 of the main contest introduced new invalidation system and additional condition: liquidations must close maximum `closable` amount, which is the amount which can be maximally closed based on the latest settled position.

The problem is that `MultiInvoker` incorrectly calculates `closableAmount` (it's not initialized and thus will often return 0 instead of correct magnitude) and thus most `LIQUIDATION` actions will revert.

Vulnerability Detail

`MultiInvoker` calculates the `closable` amount in its `_latest` function incorrectly. In particular, it doesn't initialize `closableAmount`, so it's set to 0 initially. It then scans pending positions, settling those which should be settled, and reducing `closableAmount` if necessary for remaining pending positions:

```
function _latest(
    IMarket market,
    address account
) internal view returns (Position memory latestPosition, Fixed6 latestPrice,
↪   UFixed6 closableAmount) {
    // load parameters from the market
    IPayoffProvider payoff = market.payoff();

    // load latest settled position and price
    uint256 latestTimestamp = market.oracle().latest().timestamp;
    latestPosition = market.positions(account);
    latestPrice = market.global().latestPrice;
    UFixed6 previousMagnitude = latestPosition.magnitude();

    // @audit-issue Should add:
    // closableAmount = previousMagnitude;
    // otherwise if no position is settled in the following loop, closableAmount
↪   incorrectly remains 0

    // scan pending position for any ready-to-be-settled positions
    Local memory local = market.locals(account);
    for (uint256 id = local.latestId + 1; id <= local.currentId; id++) {
```

```
        // load pending position
        Position memory pendingPosition = market.pendingPositions(account, id);
        pendingPosition.adjust(latestPosition);

        // load oracle version for that position
        OracleVersion memory oracleVersion =
↪   market.oracle().at(pendingPosition.timestamp);
        if (address(payoff) != address(0)) oracleVersion.price =
↪   payoff.payoff(oracleVersion.price);

        // virtual settlement
        if (pendingPosition.timestamp <= latestTimestamp) {
            if (!oracleVersion.valid) latestPosition.invalidate(pendingPosition);
            latestPosition.update(pendingPosition);
            if (oracleVersion.valid) latestPrice = oracleVersion.price;

            previousMagnitude = latestPosition.magnitude();
@@          closableAmount = previousMagnitude;

        // process pending positions
        } else {
            closableAmount = closableAmount
                .sub(previousMagnitude.sub(pendingPosition.magnitude().min(previ
↪   ousMagnitude)));
            previousMagnitude = latestPosition.magnitude();
        }
    }
}
```

Notice, that `closableAmount` is initialized to `previousMagnitude` **only if there is at least one position that needs to be settled**. However, if `local.latestId == local.currentId` (which is the case for most of the liquidations - position becomes liquidatable due to price changes without any pending positions created by the user), this loop is skipped entirely, never setting `closableAmount`, so it's incorrectly returned as 0, although it's not 0 (it should be the latest settled position magnitude).

Since `LIQUIDATE` action of `MultiInvoker` uses `_latest` to calculate `closableAmount` and `liquidationFee`, these values will be calculated incorrectly and will revert when trying to update the market. See the `_liquidate` market update reducing `currentPosition` by `closable` (which is 0 when it must be bigger):

```
market.update(
    account,
    currentPosition.maker.isZero() ? UFixed6Lib.ZERO :
↪   currentPosition.maker.sub(closable),
    currentPosition.long.isZero() ? UFixed6Lib.ZERO :
↪   currentPosition.long.sub(closable),
```

```
    currentPosition.short.isZero() ? UFixed6Lib.ZERO :
↪   currentPosition.short.sub(closable),
    Fixed6Lib.from(-1, liquidationFee),
    true
);
```

This line will revert because `Market._invariant` verifies that `closableAmount` must be 0 after updating liquidated position:

```
if (protected && (
@@ !closableAmount.isZero() ||
    context.latestPosition.local.maintained(
        context.latestVersion,
        context.riskParameter,
        collateralAfterFees.sub(collateral)
    ) ||
    collateral.lt(Fixed6Lib.from(-1, _liquidationFee(context, newOrder)))
)) revert MarketInvalidProtectionError();
```

Impact

All `MultiInvoker` liquidation actions will revert if trying to liquidate users without positions which can be settled, which can happen in 2 cases:

1. Liquidated user doesn't have any pending positions at all (`local.latestId == local.currentId`). This is the most common case (price has changed and user is liquidated without doing any actions) and we can reasonably expect that this will be the case for at least 50% of liquidations (probably more, like 80-90%).
2. Liquidated user does have pending positions, but no pending position is ready to be settled yet. For example, if liquidator commits unrequested oracle version which liquidates user, even if the user already has pending position (but which is not yet ready to be settled).

Since this breaks important `MultiInvoker` functionality in most cases and causes loss of funds to liquidator (revert instead of getting liquidation fee), I believe this should be High severity.

Code Snippet

There is no initialization of `closableAmount` in `MultiInvoker._latest` before the pending positions loop:
https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L361-L375

Initialization only happens when settling position: https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L393

However, the loop will often be skipped entirely if there are no pending positions at all, thus `closableAmount` will be returned uninitialized (0): https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L376

Tool used

Manual Review

Recommendation

Initialize `closableAmount` to `previousMagnitude`:

```
    function _latest(
        IMarket market,
        address account
    ) internal view returns (Position memory latestPosition, Fixed6 latestPrice,
↪   UFixed6 closableAmount) {
        // load parameters from the market
        IPayoffProvider payoff = market.payoff();

        // load latest settled position and price
        uint256 latestTimestamp = market.oracle().latest().timestamp;
        latestPosition = market.positions(account);
        latestPrice = market.global().latestPrice;
        UFixed6 previousMagnitude = latestPosition.magnitude();
+       closableAmount = previousMagnitude;
```

Discussion

sherlock-admin

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

polarzero commented:

> High. This would both cause a loss of funds and a malfunction in the protocol.

kbrizzle

Since standalone `settle` was dropped in v2, I believe it's actually impossible for a local account to have no pending positions once initialized, since a new pending position is always created at `current` when settling `latest` and `current != latest`.

Nonetheless we've fixed this here to remain consistent with the implementation in `Market`, which provides better safety in case we ever decide to bring back a standalone `settle` functionality in future versions.

panprog

> Since standalone `settle` was dropped in v2, I believe it's actually impossible for a local account to have no pending positions once initialized, since a new pending position is always created at `current` when settling `latest` and `current != latest`.

Agree, didn't think about it but indeed it's not possible to make them equal. Still, it can happen as described in point 2 in the report: if liquidator commits oracle unrequested (so the latest is before the first position settlement of the account), then the loop will never enter the "virtual settlement" part and `closableAmount` will remain 0.

Since it can still happen but only in certain edge case, this should be downgraded to medium.

panprog

Fixed

Issue M-5: MultiInvoker liquidation action will revert due to incorrect closable amount calculation for invalid oracle versions

Source: https://github.com/sherlock-audit/2023-09-perennial-judging/issues/45

Found by

panprog

The fix to issue 49 of the main contest introduced new invalidation system and additional condition: liquidations must close maximum `closable` amount, which is the amount which can be maximally closed based on the latest settled position.

The problem is that `MultiInvoker` incorrectly calculates `closableAmount` when settling invalid oracle positions and thus `LIQUIDATION` actions will revert in these cases.

Vulnerability Detail

`MultiInvoker` calculates the `closable` amount in its `_latest` function. This function basically repeats the logic of `Market._settle`, but fails to repeat it correctly for the invalid oracle version settlement. When invalid oracle version is settled, `latestPosition` invalidation should increment, but the `latestPosition` should remain the same. This is achieved in the `Market._processPositionLocal` by adjusting `newPosition` after invalidation before the `latestPosition` is set to `newPosition`:

```
if (!version.valid) context.latestPosition.local.invalidate(newPosition);
newPosition.adjust(context.latestPosition.local);
...
context.latestPosition.local.update(newPosition);
```

However, `MultiInvoker` doesn't adjust the new position and simply sets `latestPosition` to new position both when oracle is valid or invalid:

```
if (!oracleVersion.valid) latestPosition.invalidate(pendingPosition);
latestPosition.update(pendingPosition);
```

This leads to incorrect value of `closableAmount` afterwards:

```
previousMagnitude = latestPosition.magnitude();
closableAmount = previousMagnitude;
```

For example, if `latestPosition.market = 10`, `pendingPosition.market = 0` and pendingPosition has invalid oracle, then:

- `Market` will invalidate (`latestPosition.invalidation.market = 10`), adjust (`pendingPosition.market = 10`), set `latestPosition` to new `pendingPosition` (`latestPosition.maker = pendingPosition.maker = 10`), so `latestPosition.maker` correctly remains 10.
- `MultiInvoker` will invalidate (`latestPosition.invalidation.market = 10`), and immediately set `latestPosition` to `pendingPosition` (`latestPosition.maker = pendingPosition.maker = 0`), so `latestPosition.maker` is set to 0 incorrectly.

Since `LIQUIDATE` action of `MultiInvoker` uses `_latest` to calculate `closableAmount` and `liquidationFee`, these values will be calculated incorrectly and will revert when trying to update the market. See the `_liquidate` market update reducing `currentPosition` by `closable` (which is 0 when it must be bigger):

```
market.update(
    account,
    currentPosition.maker.isZero() ? UFixed6Lib.ZERO :
↪   currentPosition.maker.sub(closable),
    currentPosition.long.isZero() ? UFixed6Lib.ZERO :
↪   currentPosition.long.sub(closable),
    currentPosition.short.isZero() ? UFixed6Lib.ZERO :
↪   currentPosition.short.sub(closable),
    Fixed6Lib.from(-1, liquidationFee),
    true
);
```

This line will revert because `Market._invariant` verifies that `closableAmount` must be 0 after updating liquidated position:

```
if (protected && (
@@@ !closableAmount.isZero() ||
    context.latestPosition.local.maintained(
        context.latestVersion,
        context.riskParameter,
        collateralAfterFees.sub(collateral)
    ) ||
    collateral.lt(Fixed6Lib.from(-1, _liquidationFee(context, newOrder)))
)) revert MarketInvalidProtectionError();
```

Impact

If there is an invalid oracle version during pending position settlement in `MultiInvoker` liquidation action, it will incorrectly revert and will cause loss of funds for the liquidator who should have received liquidation fee, but reverts instead.

Since this breaks important `MultiInvoker` functionality in some rare edge cases (invalid oracle version, user has unsettled position which should settle during user liquidation with `LIQUIDATION` action of `MultiInvoker`), this should be a valid medium finding.

Code Snippet

Latest position is calculated incorrectly in `MultiInvoker`: https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L388-L389

Tool used

Manual Review

Recommendation

Both `Market` and `MultiInvoker` handle position settlement for invalid oracle versions incorrectly (`Market` issue with this was reported separately as it's completely different), so both should be fixed and the fix of this one will depend on how the `Market` bug is fixed. The way it is, `MultiInvoker` correctly adjusts pending position before invalidating `latestPosition` (which `Market` fails to do), however after such action `pendingPosition` must not be adjusted, because it was already adjusted and new adjustment should only change it by the difference from the last invalidation. The easier solution would be just not to change `latestPosition` in case of invalid oracle version, so the fix might be like this (just add `else`):

```
if (!oracleVersion.valid) latestPosition.invalidate(pendingPosition);
else latestPosition.update(pendingPosition);
```

However, if the `Market` bug is fixed the way I proposed it (by changing `invalidate` function to take into account difference in invalidation of `latestPosition` and `pendingPosition`), then this fix will still be incorrect, because invalidate will expect unadjusted `pendingPosition`, so in this case `pendingPosition` should not be adjusted after loading it, but it will have to be adjusted for positions not yet settled. So the fix might look like this:

```
    Position memory pendingPosition = market.pendingPositions(account, id);
-   pendingPosition.adjust(latestPosition);

    // load oracle version for that position
    OracleVersion memory oracleVersion =
↪   market.oracle().at(pendingPosition.timestamp);
    if (address(payoff) != address(0)) oracleVersion.price =
↪   payoff.payoff(oracleVersion.price);
```

```
    // virtual settlement
    if (pendingPosition.timestamp <= latestTimestamp) {
        if (!oracleVersion.valid) latestPosition.invalidate(pendingPosition);
-        latestPosition.update(pendingPosition);
+       else {
+           pendingPosition.adjust(latestPosition);
+           latestPosition.update(pendingPosition);
+       }
        if (oracleVersion.valid) latestPrice = oracleVersion.price;

        previousMagnitude = latestPosition.magnitude();
        closableAmount = previousMagnitude;

    // process pending positions
    } else {
+       pendingPosition.adjust(latestPosition);
        closableAmount = closableAmount
            .sub(previousMagnitude.sub(pendingPosition.magnitude().min(previousM
↪   agnitude)));
        previousMagnitude = latestPosition.magnitude();
    }
```

Discussion

sherlock-admin

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

polarzero commented:

> Medium. Perfectly explained and demonstrated in the report.

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/103

panprog

Fixed

Issue M-6: Invalid oracle version can cause the vault to open too large and risky position and get liquidated due to using unadjusted global current position

Source: https://github.com/sherlock-audit/2023-09-perennial-judging/issues/55

Found by

panprog

The fix to issue 49 of the main contest introduced new invalidation system, which stores invalidation accumulator for all positions. This means that `market.pendingPosition()` returns unadjusted global position which might be completely wrong.

The problem is that `Vault` (`StrategyLib`) uses `market.pendingPosition(global.currentId)` without adjusting it, which leads to incorrect current global position right after invalid oracle version (which creates different invalidation values for latest and current positions). This incorrect global position can lead to inflated position limits enforced in the market and vault opening too large risky position with very high leverage, which might liquidate the vault leading to loss of funds for vault users.

Vulnerability Detail

`StrategyLib._loadContext` for the market loads `currentPosition` as:

```
context.currentPosition = registration.market.pendingPosition(global.currentId);
```

However, this is unadjusted position, so its value is incorrect if invalid oracle version happens while this position is pending.

Later on, when calculating minimum and maxmium positions enforced by the vault in the market, they're calculated in `_positionLimit`:

```
function _positionLimit(MarketContext memory context) private pure returns
↪  (UFixed6, UFixed6) {
    return (
        // minimum position size before crossing the net position
        context.currentAccountPosition.maker.sub(
            context.currentPosition.maker

↪  .sub(context.currentPosition.net().min(context.currentPosition.maker))
                .min(context.currentAccountPosition.maker)
                .min(context.closable)
        ),
```

```
        // maximum position size before crossing the maker limit
        context.currentAccountPosition.maker.add(
            context.riskParameter.makerLimit
                .sub(context.currentPosition.maker.min(context.riskParameter.makerLimit))
        )
    );
}
```

And the target maker size for the market is set in `allocate`:

```
(targets[marketId].collateral, targets[marketId].position) = (
    Fixed6Lib.from(_locals.marketCollateral).sub(contexts[marketId].local.collateral),
    _locals.marketAssets
        .muldiv(registrations[marketId].leverage, contexts[marketId].latestPrice.abs())
        .min(_locals.maxPosition)
        .max(_locals.minPosition)
);
```

Since `context.currentPosition` is incorrect, it can happen that both `_locals.minPosition` and `_locals.maxPosition` are too high, the vault will open too large and risky position, breaking its risk limit and possibly getting liquidated, especially if it happens during high volatility.

Impact

If invalid oracle version happens, the vault might open too large and risky position in such market, potentially getting liquidated and vault users losing funds due to this liquidation.

Code Snippet

`StrategyLib._loadContext` loads current global position without adjusting it, meaning `context.currentPosition` is incorrect if invalid oracle version happens: https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L131

This leads to incorrect position limit calculations: https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L172-L187

This, in turn, leads to incorrect target vault's position calculation for the market: https://github.com/sherlock-audit/2023-09-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/lib/StrategyLib.sol#L93-L99

Tool used

Manual Review

Recommendation

Adjust global current position after loading it:

```
    context.currentPosition =
↪   registration.market.pendingPosition(global.currentId);
+   context.currentPosition.adjust(registration.market.position());
```

Discussion

kbrizzle

Fixed in: https://github.com/equilibria-xyz/perennial-v2/pull/109.

Please note there were additional pending positions that required adjustment.

panprog

Fixed

Issue M-7: [Perennial Self Review] Incorrect price used during liquidation calculation

Source: https://github.com/sherlock-audit/2023-09-perennial-judging/issues/59

Found by

Protocol Team Fixed by https://github.com/equilibria-xyz/perennial-v2/pull/108

Discussion

panprog

Fixed but when combined with #9, the liquidation will revert due to the fix (#9 will make oracle.latest().price = 0 and this will be used by the fix, while the correct `Market` process will use the latest valid oracle price in this case). The probability of this situation is very low, but still possible.

kbrizzle

Noted: we made this intentional decision since liquidations are generally predicated on posting a valid non-requested price anyways.

jacksanford1

Based on @kbrizzle's comment, Sherlock will consider this issue as acknowledged due to a very low but still possible potential for reverting.