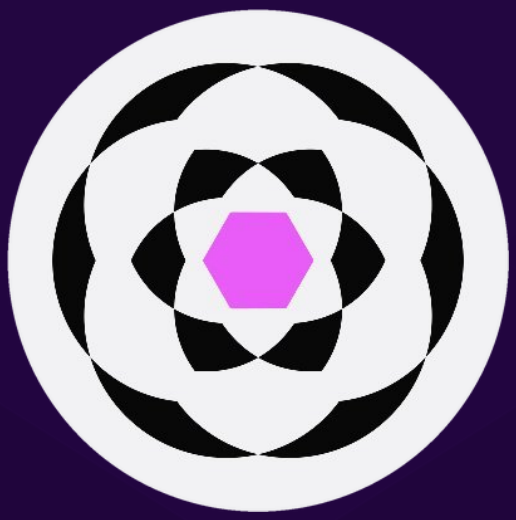
SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for:	Perennial
Prepared by:	Sherlock
Lead Security Expert:	WATCHPUG
Dates Audited:	July 24 - August 15, 2023
Prepared on:	September 18, 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/perennial-v2

Branch: main

Commit: 0937c269120b2c5383c883da547f080ce9dc6cca

Repository: equilibria-xyz/root

Branch: v2

Commit: 74fdd3e1acbca319f6825f44a971c8c196ebc2a9

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
11	6

Security experts who found valid issues

panprog
KingNFT
WATCHPUG
Emmanuel
bin2chen
minhtrng
0x73696d616f
Vagner
n33k
moneyversed

Issue H-1: Oracle request timestamp and pending position timestamp mismatch can make most position updates invalid

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/42

Found by

KingNFT, WATCHPUG, minhtrng, panprog

When a new pending position is added, its timestamp is set to `currentTimestamp` returned by oracle's `status` function, which is a timestamp at certain granularities rounding up into the future, which means that most of the time it's greater than `block.timestamp`. However, when `request` is called for the oracle, the request timestamp is set to `block.timestamp`. Due to this mismatch, when the oracle price is commited, it is commited with request's timestamp, but when the position is settled, it tries to read the price at position's timestamp, which is a different time. As such, if the oracle price is commited for each request, it's still easily possible that all pending positions will have invalid oracle versions, completely breaking the protocol's functionality.

Vulnerability Detail

An example of what happens exactly:

1. PythOracle granularity is set to 100.
2. User opens position at timestamp = 101. Pending position is stored with timestamp = 200 (because PythOracle returns currentTimestamp = 200)
3. At the same time `oracle.request()` is called, which stores 101 (current timestamp) into `versionList`
4. User calls `oracle.commitRequested()`, which stores current price into `_prices[101]`
5. Later when that pending position is settled, it requests `oracle.at(200)` which doesn't have a price set (is invalid).

The same will happen to all pending positions - so most of them will easily be invalid, which will completely break the protocol and cause all kinds of problems due to pending positions being invalid and not updating profit and loss properly.

Impact

The most straightforward impact is unexpectedly long position commit times and possible funds loss due to this, if the oracle commit flow is the normal expected

flow (only commit requested versions). For example: T=1: User A requests to open position long = 1. Position timestamp = 100. Oracle request timestamp = 1 T=15: Oracle commits requested version at timestamp = 1, price = $100. T=10010: User B requests to open position. Position timestamp = 10100. Oracle request timestamp = 10010 T=10025: Oracle commits requested version at timestamp = 10010, price = $110.

User A expects to be filled at price close to $100. However, he's only filled when the next user trades after him, which happens much later than expected with a very different price ($110), so User A has lost $10 unexpectedly. Basically, each user will only be settled when the next user trades. In quiet markets this can lead to very long settlement times and very bad prices for users.

User A, however, can notice these long waiting times and can fix it by voluntary commiting non-requested versions. For example, he can commit at T=120 and be filled with the correct price. However, this will mean that all commits must be made non-requested, thus they will not be rewarded with the keeper fees. So the user will pay keeper fees when trading, but will also be forced to lose gas fees for oracle commits, so either broken and long waiting times, or broken oracle non-rewarded updates: both are high impacts.

Another impact is completely broken internal accounting due to a lot of invalid oracle versions. There is a different bug reported by me about desync of global and local positions during invalid oracles. This bug, when coupled with the desync of global and local positions, will lead to catastrophic consequences and complete breakage of accounting of collateral, bank run and loss of funds for users. Scenario of what can (and will) happen: User B has active open position maker=2 with collateral = 100 T=99: User A opens long=1 with collateral=100: update(0,1,0,100) (pending position timestamp = 100) T=101: User A decides to close: update(0,0,0,0) (pending position timestamp = 200) T=130: Oracle commited for timestamp=110, price = $100 (user A position at timestamp = 100 is invalid) T=150: User B settles: update(2,0,0,0) T=220: Oracle commited for timestamp=205, price = $90 after settlement of user A and user B: user A will have collateral = $100 (local pending position long = 1 at timestamp = 100 will be invalidated and ignored) user B will have collateral = $110 (global pending position long = 1 will be current at timestamp 110 and accumulate pnl from timestamp 110 to timestamp=205)

So total deposit of both users is $100 + $100 = $200 Total collateral in the end: $100 + $110 = $210 But protocol only has $200 in funds, so users will be unable to withdraw everything, which can cause bank run and loss of funds for the last user.

Such situations will happen all the time by themselves due to lots of invalid oracle versions, so this will mess up accounting completely.

For the details of this bug, you can refer to my other report.

Code Snippet

1. Oracle `status()` returns timestamp which is in the future.

Oracle `status()` returns timestamp directly from current provider's `status()`: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/Oracle.sol#L47

PythOracle `status()` timestamp is taken from `current()`, which in turn returns `current()` from PythFactory: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L101

PythFactory `current()` returns timestamp which is granulated into the future using ceilDiv, which rounds up: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythFactory.sol#L76

2. Pending position's timestamp is taken from oracle `status()`.

`context.currentTimestamp` is set to timestamp from `oracle.status()`: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L312 https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L575

New pending positions (global and local) timestamp is set to `context.currentTimestamp`: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L267-L269

And `request()` from oracle is done at the same time: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L284

3. PythOracle `request()` stores `block.timestamp` in the request list (called `versionList`) (**not** `current()` timestamp): https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L77-L81

4. PythOracle `commitRequested()` sets price at `versionList` timestamp (i.e. `block.timestamp` at the time `request()` was made)

`versionToCommit` is stored request's timestamp: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L135

The commit price is stored at the `versionToCommit` timestamp: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L154 https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L202-L203

Tool used

Manual Review

Recommendation

Make timestamp of pending positions and timestamp of oracle request match. Record `current()` as a timestamp for the `request()`:

```
function request(address) external onlyAuthorized {
    uint nextTimestamp = current();
    if (versionList.length == 0 || versionList[versionList.length - 1] <
↪   nextTimestamp) {
        versionList.push(nextTimestamp);
    }
}
```

Discussion

sherlock-admin

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

141345 commented:

> h

arjun-io

Fixed via: https://github.com/equilibria-xyz/perennial-v2/pull/57

jacksanford1

From WatchPug:

Fixed.

With the updated code, when there are multiple new orders within the same hour, while the keeper only gets 1 unit of the keeper fee, each order will pay for 1 unit of keeper fee.

A more fair approach would be either:

Only the first request pays the settlement/keeper fee. Splitting the settlement fee among the orders.

arjun-io

> From WatchPug:
>
> Fixed.

> With the updated code, when there are multiple new orders within the same hour, while the keeper only gets 1 unit of the keeper fee, each order will pay for 1 unit of keeper fee.
>
> A more fair approach would be either:
>
> Only the first request pays the settlement/keeper fee. Splitting the settlement fee among the orders.

While true, we choose specifically not to go the route of the suggestion due to complexity versus benefit.

Oracle versions (granularity) in V2 are on the order of 5-30 seconds, so it's less likely that many orders will happen in the same market at the same version. To add this, we'd either have to only charge the first request of a version, or pro-rate the amount charged retroactively – either adding significant code complexity, or weird economic situations. Finally, we have plans for future updates which will incur per-request cost (settling a user's account for them when the keeper posts the new price) so we'd like to keep this open.

Issue H-2: Invalid oracle versions can cause desync of global and local positions making protocol lose funds and being unable to pay back all users

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/49

Found by

panprog

When oracle version is skipped for any reason (marked as invalid), pending positions are invalidated (reset to previous latest position):

```
    function _processPositionGlobal(Context memory context, uint256
↪   newPositionId, Position memory newPosition) private {
        Version memory version =
↪   _versions[context.latestPosition.global.timestamp].read();
        OracleVersion memory oracleVersion = _oracleVersionAtPosition(context,
↪   newPosition);
>         if (!oracleVersion.valid)
↪   newPosition.invalidate(context.latestPosition.global);
...
    function _processPositionLocal(
        Context memory context,
        address account,
        uint256 newPositionId,
        Position memory newPosition
    ) private {
        Version memory version = _versions[newPosition.timestamp].read();
>         if (!version.valid)
↪   newPosition.invalidate(context.latestPosition.local);
```

This invalidation is only temporary, until the next valid oracle version. The problem is that global and local positions can be settled with different next valid oracle version, leading to temporary desync of global and local positions, which in turn leads to incorrect accumulation of protocol values, mostly in profit and loss accumulation, breaking internal accounting: total collateral of all users can increase or decrease due to this while the funds deposited remain the same, possibly triggering a bank run, since the last user to withdraw will be unable to do so, or some users might get collateral reduced when it shouldn't (loss of funds for them).

Vulnerability Detail

In more details, if there are 2 pending positions with timestamps different by 2 oracle versions and the first of them has invalid oracle version at its timestamp,

then there are 2 different position flows possible depending on the time when the position is settled (update transaction called):

1. For earlier update the flow is: previous position (oracle v1) -> position 1 (oracle v2) -> position 2 (oracle v3)
2. For later update position 1 is skipped completely (the fees for the position are also not taken) and the flow is: previous position (oracle v1) -> invalidated position 1 (in the other words: previous position again) (oracle v2) -> position 2 (oracle v3)

While the end result (position 2) is the same, it's possible that pending global position is updated earlier (goes the 1st path), while the local position is updated later (goes the 2nd path). For a short time (between oracle versions 2 and 3), the global position will accumulate everything (including profit and loss) using the pending position 1 long/short/maker values, but local position will accumulate everything using the previous position with different values.

Consider the following scenario: Oracle uses granularity = 100. Initially user B opens position maker = 2 with collateral = 100. T=99: User A opens long = 1 with collateral = 100 (pending position long=1 timestamp=100) T=100: Oracle fails to commit this version, thus it becomes invalid T=201: At this point oracle version at timestamp 200 is not yet commited, but the new positions are added with the next timestamp = 300: User A closes his long position (update(0,0,0,0)) (pending position: long=1 timestamp=100; long=0 timestamp=300) At this point, current global long position is still 0 (pending the same as user A local pending positions)

T=215: Oracle commits version with timestamp = 200, price = $100 T=220: User B settles (update(2,0,0,0) - keeping the same position). At this point the latest oracle version is the one at timestamp = 200, so this update triggers update of global pending positions, and current latest global position is now long = 1.0 at timestamp = 200. T=315: Oracle commits version with timestamp = 300, price = $90 after settlement of both UserA and UserB, we have the following:

1. Global position settlement. It accumulates position [maker = 2.0, long = 1.0] from timestamp = 200 (price=$100) to timestamp = 300 (price=$90). In particular: longPnl = 1*($90-$100) = -$10 makerPnl = -longPnl = +$10
2. User B local position settlement. It accumulates position [maker = 2.0] from timestamp = 200 to timestamp = 300, adding makerPnl ($10) to user B collateral. So user B collateral = $110
3. User A local position settlement. When accumulating, pending position 1 (long = 1, timestamp = 100) is invalidated to previous position (long = 0) and also fees are set to 0 by invalidation. So user A local accumulates position [long = 0] from timestamp = 0 to timestamp = 300 (next pending position), this doesn't change collateral at all (remains $100). Then the next pending position [long = 0] becomes the latest position (basically position of long=1 was completely ignored as if it has not existed).

Result: User A deposited $100, User B deposited $100 (total $200 deposited) after the scenario above: User A has collateral $110, User B has collateral $100 (total $210 collateral withdrawable) However, protocol only has $200 deposited. This means that the last user will be unable to withdraw the last $10 since protocol doesn't have it, leading to a user loss of funds.

Impact

Any time the oracle skips a version (invalid version), it's likely that global and local positions for different users who try to trade during this time will desync, leading to messed up accounting and loss of funds for users or protocol, potentially triggering a bank run with the last user being unable to withdraw all funds.

The severity of this issue is high, because while invalid versions are normally a rare event, however in the current state of the codebase there is a bug that pyth oracle requests are done using this block timestamp instead of granulated future time (as positions do), which leads to invalid oracle versions almost for all updates (that bug is reported separately). Due to this other bug, the situation described in this issue will arise very often by itself in a normal flow of the user requests, so it's almost 100% that internal accounting for any semi-active market will be broken and total user collateral will deviate away from real deposited funds, meaning the user funds loss.

But even with that other bug fixed, the invalid oracle version is a normal protocol event and even 1 such event might be enough to break internal market accounting.

Proof of concept

The scenario above is demonstrated in the test, add this to test/unit/market/Market.test.ts:

```
it('panprog global-local desync', async () => {
    const positionMaker = parse6decimal('2.000')
    const positionLong = parse6decimal('1.000')
    const collateral = parse6decimal('100')

    const oracleVersion = {
        price: parse6decimal('100'),
        timestamp: TIMESTAMP,
        valid: true,
    }
    oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
    oracle.status.returns([oracleVersion, oracleVersion.timestamp + 100])
    oracle.request.returns()

    dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪   collateral.mul(1e12)).returns(true)
```

```
    await market.connect(userB).update(userB.address, positionMaker, 0, 0,
↪   collateral, false)

    const oracleVersion2 = {
        price: parse6decimal('100'),
        timestamp: TIMESTAMP + 100,
        valid: true,
    }
    oracle.at.whenCalledWith(oracleVersion2.timestamp).returns(oracleVersion2)
    oracle.status.returns([oracleVersion2, oracleVersion2.timestamp + 100])
    oracle.request.returns()

    dsu.transferFrom.whenCalledWith(user.address, market.address,
↪   collateral.mul(1e12)).returns(true)
    await market.connect(user).update(user.address, 0, positionLong, 0,
↪   collateral, false)

    var info = await market.locals(userB.address);
    console.log("collateral deposit maker: " + info.collateral);
    var info = await market.locals(user.address);
    console.log("collateral deposit long: " + info.collateral);

    // invalid oracle version
    const oracleVersion3 = {
        price: 0,
        timestamp: TIMESTAMP + 200,
        valid: false,
    }
    oracle.at.whenCalledWith(oracleVersion3.timestamp).returns(oracleVersion3)

    // next oracle version is valid
    const oracleVersion4 = {
        price: parse6decimal('100'),
        timestamp: TIMESTAMP + 300,
        valid: true,
    }
    oracle.at.whenCalledWith(oracleVersion4.timestamp).returns(oracleVersion4)

    // still returns oracleVersion2, because nothing commited for version 3, and
↪   version 4 time has passed but not yet commited
    oracle.status.returns([oracleVersion2, oracleVersion4.timestamp + 100])
    oracle.request.returns()

    // reset to 0
    await market.connect(user).update(user.address, 0, 0, 0, 0, false)

    // oracleVersion4 commited
    oracle.status.returns([oracleVersion4, oracleVersion4.timestamp + 100])
```

```
    oracle.request.returns()

    // settle
    await market.connect(userB).update(userB.address, positionMaker, 0, 0, 0,
↪   false)

    const oracleVersion5 = {
        price: parse6decimal('90'),
        timestamp: TIMESTAMP + 400,
        valid: true,
    }
    oracle.at.whenCalledWith(oracleVersion5.timestamp).returns(oracleVersion5)
    oracle.status.returns([oracleVersion5, oracleVersion5.timestamp + 100])
    oracle.request.returns()

    // settle
    await market.connect(userB).update(userB.address, positionMaker, 0, 0, 0,
↪   false)
    await market.connect(user).update(user.address, 0, 0, 0, 0, false)

    var info = await market.locals(userB.address);
    console.log("collateral maker: " + info.collateral);
    var info = await market.locals(user.address);
    console.log("collateral long: " + info.collateral);
})
```

Console output for the code:

```
collateral deposit maker: 100000000
collateral deposit long: 100000000
collateral maker: 110000028
collateral long: 100000000
```

Maker has a bit more than $110 in the end, because he also earns funding and interest during the short time when ephemeral long position is active (but user A doesn't pay these fees).

Code Snippet

`_processPositionGlobal` invalidates position if oracle version is invalid for its timestamp: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L390-L393

`_processPositionLocal` does the same: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L430-L437

`_settle` loops over global and local positions until the latest oracle version timestamp. In this loop each position is invalidated to previous latest if it has invalid oracle timestamp. So if `_settle` is called after the invalid timestamp, previous latest is accumulated for it: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L333-L347

Later in the `_settle`, the latest global and local position are advanced to latestVersion timestamp, the difference from the loop is that since position timestamp is set to valid oracle version, `_processPositionGlobal` and `_processPositionLocal` here will be called with valid oracle and thus position (which is otherwise invalidated in the loop) will be valid and set as the latest position: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L349-L360

This means that for early timestamps, invalid version positions will become valid in the `sync` part of the `_settle`. But for late timestamps, invalid version position will be skipped completely in the loop before `sync`. This is the core reason of desync between local and global positions.

Tool used

Manual Review

Recommendation

The issue is that positions with invalid oracle versions are ignored until the first valid oracle version, however the first valid version can be different for global and local positions. One of the solutions I see is to introduce a map of position timestamp -> oracle version to settle, which will be filled by global position processing. Local position processing will follow the same path as global using this map, which should eliminate possibility of different paths for global and local positions.

It might seem that the issue can only happen with exactly 1 oracle version between invalid and valid positions. However, it's also possible that some non-requested oracle versions are commited (at some random timestamps between normal oracle versions) and global position will go via the route like t100[pos0]->t125[pos1]->t144[pos1]->t200[pos2] while local one will go t100[pos0]->t200[pos2] OR it can also go straight to t300 instead of t200 etc. So the exact route can be anything, and local oracle will have to follow it, that's why I suggest a path map.

There might be some other solutions possible.

Discussion

sherlock-admin

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

141345 commented:

> m

panprog

Escalate

This should be high, because:

1. When the situation described in the issue happens, it causes serious internal accounting issue causing loss of funds by users and possibly causing bank run and then loss of funds by the last users.
2. The condition for this to happen is invalid oracle version, which in the current state of the code will happen regularly (see #42)
3. For the situation described to happen it's enough for 1 user to request to open position, oracle to be invalid for that position timestamp, and then the user to request any modification to this position (increase or decrease), then another user to do any action after the next oracle is commited. That's it, internal accounting is broken.
4. The stated flow of events can happen by itself very regularly in any semi-active market, leading to worse and worse accounting broking up.
5. **OR** malicious user can try to abuse this scenario to profit off it or just to break the protocol. Doing this is mostly free (except for some keeper fees). In such case the protocol will be broken very quickly.

I don't know why it's judged medium, but this issue is very likely to happen and will cause a lot of damage to the market, thus it should be high.

sherlock-admin2

> Escalate
>
> This should be high, because:
>
> 1. When the situation described in the issue happens, it causes serious internal accounting issue causing loss of funds by users and possibly causing bank run and then loss of funds by the last users.
> 2. The condition for this to happen is invalid oracle version, which in the current state of the code will happen regularly (see #42)
> 3. For the situation described to happen it's enough for 1 user to request to open position, oracle to be invalid for that position timestamp, and then the user to request any modification to this position (increase or decrease), then another user to do any action

after the next oracle is commited. That's it, internal accounting is broken.

4. The stated flow of events can happen by itself very regularly in any semi-active market, leading to worse and worse accounting broking up.
5. **OR** malicious user can try to abuse this scenario to profit off it or just to break the protocol. Doing this is mostly free (except for some keeper fees). In such case the protocol will be broken very quickly.

> I don't know why it's judged medium, but this issue is very likely to happen and will cause a lot of damage to the market, thus it should be high.

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.

Emedudu

Escalate

> When oracle version is skipped for any reason (marked as invalid), pending positions are invalidated (reset to previous latest position):

This is not HIGH because there is a limitation: "When oracle version is skipped for any reason" This is a VERY unlikely event. So by Sherlock rules, it is a MEDIUM: "Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost."

panprog

> This is not HIGH because there is a limitation: "When oracle version is skipped for any reason" This is a VERY unlikely event.

While it is supposed to be a rare event, in the current state of the code, this is a **VERY LIKELY** event, see #42 It should be judged based on the current code, not on assumptions of how it will work in the future.

Emedudu

Skipping of oracle versions is an unlikely event. This was stated under the impact section of the issue: "The severity of this issue is high, `because while invalid versions are normally a rare event`, however in the current state of the codebase there is a bug that pyth oracle requests are done using this block timestamp instead of granulated future time (as positions do), which leads to invalid oracle versions almost for all updates (that bug is reported separately)."

This makes this issue to fall under MEDIUM severity according to Sherlock's classification rules.

Minh-Trng

> It should be judged based on the current code, not on assumptions of how it will work in the future.

it should be treated like different submissions describing different impacts with the same root cause: does fixing that one root cause mitigate all described impacts? then all of them are considered duplicates.

now, clearly your submission is not a duplicate, but it builds on that same root cause. if this root cause were fixed, your impact would still hold, but with a much lower likelihood

panprog

> now, clearly your submission is not a duplicate, but it builds on that same root cause. if this root cause were fixed, your impact would still hold, but with a much lower likelihood

Yes, I chain 2 issues to demonstrate high impact. In this case both issues should be high. We can't start predicting future "what happens if that one is fixed..." The way it is now - existance of either issue creates a high impact for the protocol, and each issue is a separate one. I disagree that the issue should be "isolated" and impact considered as if the other issues are fixed. Sherlock has the following judging rule:

> Future issues: Issues that result out of a future integration/implementation that was not intended (mentioned in the docs/README) or because of a future change in the code (as a fix to another issue) are not valid issues.

While it doesn't provide the same clear rule for the opposite, I believe it's a logical continuation of that rule to that impact shouldn't be decreased because of a future change in the code (as a fix to another issue).

sherlock-admin2

> Escalate
>
> > When oracle version is skipped for any reason (marked as invalid), pending positions are invalidated (reset to previous latest position):
>
> This is not HIGH because there is a limitation: "When oracle version is skipped for any reason" This is a VERY unlikely event. So by Sherlock rules, it is a MEDIUM: "Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost."

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.

Minh-Trng

The judging rule explicitly talks about **unintended** future implementation. So your logical continuation would also only be applicable to unintended future implementations (which I would absolutely agree with).

Your submission however even shows that you were aware of the correct **intended** future implementations and the change in likelihood that it would bring.

panprog

> The judging rule explicitly talks about **unintended** future implementation.

It says **OR**: so either unintended future implementation **OR** because of future fix of another issue.

I still think this is high because

1. Even by itself, it messes up accounting and causes loss of funds if oracle version is invalid. Invalid oracle versions are normal protocol operation, even if rare. This can be compared to liquidations - they're also rare but normal protocol operation, so issues in liquidations are considered high. Similarly, this issue due to invalid oracle version should also be high.
2. In the current implementation, oracle versions are invalid very frequently due to another bug, so this should be high either way. I still think disregarding the other bugs when considering impact is incorrect. And the fact that I'm aware of the intended future implementation is irrelevant: the way it is right now, the issue in this report happens very frequently by itself.

141345

Medium severity seems more appropriate.

Because:

- the likelihood that "oracle version is skipped" is not common scenario.
- the loss is not significant.

Based on sherlock's H/M criteria

> Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future.

panprog

- the likelihood that "oracle version is skipped" is not common scenario.

In the current state of the code it is **very likely** scenario (*expected* behavior is for this to be not common scenario, but currently it is common)

- the loss is not significant.

This depends. If malicious user want to cause damage (or comes up with a profitable scenario), the loss can be very significant. In the current state of the code - since it will happen often by itself, each instance will not be very significant (0.01-0.2% of the protocol funds depending on price volatility), but it will add up to large amounts over time.

141345

Although https://github.com/sherlock-audit/2023-07-perennial-judging/issues/42 points out the possibility of skipped version, it does not look like some common events.

0.01-0.2% each time won't be considered material loss. Some normal operation can have even more profit than that, such as spot-futures arbitrage, cross exchange arbitrage.

And the attacker need to control the conditions to perform the action. As such, the loss amount and requirements fall into the Med.

panprog

I've added a detailed response previously, but don't see it now, maybe it got deleted or not sent properly. Here it is again.

> Although #42 points out the possibility of skipped version, it does not look like some common events.

No, it's very easy to have skipped versions in the current implementation. For example: t=24: user opens postiion (request timestamp = 25, position timestamp = 100) t=48: user closes position (request timestamp = 48, position timestamp = 100) t=96: user opens position (request timestamp = 96, position timestamp = 100) t=108: user closes position (request timestamp = 108, position timestamp = 200) ... If oracle is commited requested (in the normal operation of the protocol), the commited timestamps will be: 25, 48, 96, 108. Timestamp = 100 will be missing and all position at this timestamp will have invalid oracle version. In order to have a valid oracle version for position, a request to open or close position must be made at exactly the timestamp of the previous positions (t=100 in example). However, a lot of times there won't be even a block with such timestamp. For example, in ethereum blocks are currently happening every 12 seconds and have odd timestamps, and in the other networks the time between blocks is random, so the probability to actually have the block timestamp divisible by granularity is low. Even if granularity and block timestamp align well, it still requires that request is made at

exactly the granularity timestamp, so for example if granularity = 120 and time between blocks is 12, then every 10th block must have position open in order to request at the right timestamp. It's still possible to commit unrequested, but first, this is not incentivized (as there is no reward to the keeper who commits unrequested) and second, there is still a time window when this has to be commited. So in the example above, commit unrequested for timestamp = 100 can only be done after timestamp = 96 is commited but before the timestamp = 108 is commited. So it's still easy to miss this time window to commit unrequested. So in summary, the way it is now, it's easier to have invalid oracle version, than it is to have valid oracle version.

> 0.01-0.2% each time won't be considered material loss. Some normal operation can have even more profit than that, such as spot-futures arbitrage, cross exchange arbitrage.

I argue that this is actually a material loss - it's the percentage off the **protocol funds**. So if $100M are deposited into protocol, the loss can be like $100K per instance. And since it can happen almost every granularity, this will add up very quick.

> And the attacker need to control the conditions to perform the action. As such, the loss amount and requirements.

Even if there is no attacker, the loss will be smaller, but it will be continuous in time, so even if it's, say, $10K per instance (with $100M deposited), it will add up to tens of millions over less than a day.

panprog

Regarding the impact, I want to point that #62 is high and has similar impact (messed up internal accounting), however the real damage from it is:

> The global account's assets and shares should be calculated with toAssetsGlobal and toSharesGlobal respectively, but now, they are calculated with toAssetsLocal and toSharesLocal. toAssetsGlobal subtracts the globalKeeperFees from the global deposited assets, while toAssetsLocal subtracts globalKeeperFees/Checkpoint.count fees from the local account's assets.

So the real damage from #62 is reduction of deposited assets by (keeper fees / count) instead of (keeper fees). Since keeper fees are low (comparable to gas fees), the real damage is not that large compared to funds deposited (less than 0.001% likely).

However, the issue from this report also causes messed up internal accounting, but the real damage depends on position size and price volatility and will be much higher than in #62 on average, even when happening by itself. If coming from malicious parties, this can be a very large amount.

Even though damage in #62 is much easier to inflict, I believe that due to higher

damage per instance of this issue, the overall damage over time from this issue will be higher than from #62. Something like: $5 per granilarity time from #62 (and still has to be executed by attacker - so the attack has a gas cost of similar amount) $10K-$100K per invalid oracle from this one - currently that's maybe once per 10-100 granularities by itself in the current state, so at least $100-$1000 per granularity time (and happens by itself).

So based on possible real overall damage caused, this issue should be high.

Emedudu

This should be MEDIUM because skipped oracle versions is not a common event.

> In the current state of the code it is very likely scenario (expected behavior is for this to be not common scenario, but currently it is common)

Skipped oracle versions is unlikely, and the reason why it is a likely scenario in the current code is due to a bug, which has already been reported in #42. Fixing issue 42 will cause the possibility of this to be unlikely. So, this report is a combination of issue 42(which is already of high severity), and another bug, which is very unlikely by itself.

Since different attack scenarios, with same fixes are considered duplicates, this issue should be a MEDIUM because issue 42(which allows this bug to be a likely scenario) when fixed, will make this issue unlikely

141345

1st, the scenario is conditional, not the kind on daily basis. 2nd, loss magnitude like 0.01-0.2% is common, different exchanges could have that magnitude of price difference, (common arbitrage opportunity, future contracts of different terms can also have larger arbitrage than this).

As such, conditional loss and capped loss amount, will suggest medium severity.

panprog

> 1st, the scenario is conditional, not the kind on daily basis.

I don't think high severity means unconditional. My understanding is that high severity is high impact which can happen with rather high probability. And high probability doesn't mean it can happen every transaction, it just means that it can reasonably happen within a week or a month. Example (from the other contest): liquidation can be frontrun to block it without spending funds: was judged high, even though liquidation is not very frequent (days can pass without single liquidation).

And this issue probability to happen is high.

> 2nd, loss magnitude like 0.01-0.2% is common, different exchanges could have that magnitude of price difference, (common arbitrage

> opportunity, future contracts of different terms can also have larger arbitrage than this).

Why do you only consider it as a 1 time event? The way it is now, that'll be like 0.1% per 1-10 minutes, this will add up to 10%+ in less than a day.

> As such, conditional loss and capped loss amount, will suggest medium severity.

I think there are no new arguments presented here, so I keep it up to Sherlock to decide. I think ultimately it comes down to:

- this issue is high as is now due to the other bug.
- if the other bug is fixed (and oracle versions work the way they're supposed to work), this one will be medium.

So my argument is that the way it is now, it's high and severity shouldn't be downgraded as if the other bug is fixed.

141345

I don't think it's one time, it's something intermittent, it happens every once in a while, but the frequency might not be as high as per 1-10 minutes.

And even it happened, there could be loss, and also sometimes no loss.

That's why based on the probability of happending and loss, it is more suitable for conditional and capped loss.

panprog

I still think this is High, the way it is now - it can happen as often as once per each granularity if malicious user abuses it, or maybe once per 10 granularities with semi-active trading by itself. Either way it's very possible and causes loss of funds. As I said earlier, I see this as medium only if #42 is disregarded, but I don't see any ground to ignore that bug when assessing severity of this one.

141345

What about the cross exchange price difference, the magnitude could be the same level. Also the spot and perpetual contract could deviates, those cases are not considered exchange's loss.

panprog

> What about the cross exchange price difference, the magnitude could be the same level. Also the spot and perpetual contract could deviates, those cases are not considered exchange's loss.

It's different, they're normal protocol operation and funds just changing hands. This issue leads to mismatch between funds protocol has and funds protocol owes

(protocol has 100, but total collateral users can withdraw can be 200, so not all users can withdraw, which can trigger bank run with last users unable to withdraw)

kbrizzle

To chime in here from our perspective -- this issue identified a pretty fundamental flaw in our accounting system that would have caused the markets to be completely out of sync (both w.r.t. positions as well as balances) in the event of an unfortunately timed invalid version.

While ideally rare, occasional invalid versions are expected behavior. Invalid versions can occur for a number of reasons during the course of normal operation:

- Underlying oracle does not have data available for any timestamp in the validity window (occurs in Pyth from time to time)
- No keeper responds within grace period window
- Future oracle integration / upgrade adds anomaly detection to invalidate versions via a number of metrics

Given that this bug would have likely surfaced in normal operation plus its noted effect, our opinion is that this should be marked as High.

Fixed in: https://github.com/equilibria-xyz/perennial-v2/pull/82 and https://github.com/equilibria-xyz/perennial-v2/pull/94.

hrishibhat

Result: High Unique After considering all the comments above and discussing this further. In addition to the Sponsor comments above the issue highlights the issues in context with the current state of the codebase. the submission justifies the severity by mentioning another underlying issue in the impact section clearly, which is again pointed out in the escalation. The future implementation rule does not apply here. Also, Agree with the points raised by @panprog in the subsequent comments. Considering this a valid high

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted
- Emedudu: rejected

jacksanford1

From WatchPug:

https://github.com/equilibria-xyz/perennial-v2/blob/602218ca8cb62db649bf4b6a6722824e9ab20166/packages/perennial/contracts/Market.sol#L563-L595

L590 will revert due to underflow in the following case:

context.latestPosition.local.magnitude() is equal to 0 pendingLocalPositions[0].magnitude() is equal to 10 pendingLocalPositions[1].magnitude() is equal to 5 The initial value of previousMagnitude is 0.

After the first run (i: 0):

closableAmount is equal to 0 previousMagnitude is equal to 10 At the second run (i: 1), at L590, 0 - 10 will revert.

arjun-io

> From WatchPug:
>
> https://github.com/equilibria-xyz/perennial-v2/blob/602218ca8cb62db649bf4b6a6722824e9ab20166/packages/perennial/contracts/Market.sol#L563-L595
>
> context.latestPosition.local.magnitude() is equal to 0 pendingLocalPositions[0].magnitude() is equal to 10 pendingLocalPositions[1].magnitude() is equal to 5 The initial value of previousMagnitude is 0.
>
> After the first run (i: 0):
>
> closableAmount is equal to 0 previousMagnitude is equal to 10 At the second run (i: 1), at L590, 0 - 10 will revert.

Under the new delta invalidation system, the aggregate amount of your pending closes must be less than your currently settled (latest) position as specified here. This underflow revert is actually intentional to enforce this invariant. See this test to cross-reference expected behavior.

Issue H-3: Protocol fee from Market.sol is locked

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/52

Found by

0x73696d616f, Emmanuel, WATCHPUG, bin2chen

The `MarketFactory#fund` calls the specified market's `Market#claimFee` function. This will send the protocolFee to the MarketFactory contract. MarketFactory contract does not max approve any address to spend its tokens, and there is no function that can be used to get the funds out of the contract, so the funds are permanently locked in MarketFactory.

Vulnerability Detail

Here is `MarketFactory#fund` function:

```
  function fund(IMarket market) external {
        if (!instances(IInstance(address(market)))) revert
↪   FactoryNotInstanceError();
@>      market.claimFee();
    }
```

This is `Market#claimFee` function:

```
function claimFee() external {
    Global memory newGlobal = _global.read();

    if (_claimFee(address(factory()), newGlobal.protocolFee))
↪   newGlobal.protocolFee = UFixed6Lib.ZERO;
    ...
}
```

This is the internal `_claimFee` function:

```
function _claimFee(address receiver, UFixed6 fee) private returns (bool) {
    if (msg.sender != receiver) return false;

    token.push(receiver, UFixed18Lib.from(fee));
    emit FeeClaimed(receiver, fee);
    return true;
}
```

As we can see, when `MarketFactory#fund` is called, Market#claimFee gets called which will send the protocolFee to msg.sender(MarketFacttory). When you check

through the MarketFactory contract, there is no place where another address(such as protocol multisig, treasury or an EOA) is approved to spend MarketFactory's funds, and also, there is no function in the contract that can be used to transfer MarketFactory's funds. This causes locking of the protocol fees.

Impact

Protocol fees cannot be withdrawn

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/MarketFactory.sol#L89

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L133

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L145-L151

Tool used

Manual Review

Recommendation

Consider adding a `withdraw` function that protocol can use to get the protocolFee out of the contract. You can have the withdraw function transfer the MarketFactory balance to the treasury or something.

Discussion

sherlock-admin

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

141345 commented:

> h

n33k commented:

> medium

arjun-io

We originally wanted to keep the funds in the Factory (for a future upgrade) but it might make sense to instead allow the Factory Owner (Timelock) to claim these funds instead

Emedudu

Escalate

I believe this is of HIGH severity because funds are permanently locked

sherlock-admin2

> Escalate
>
> I believe this is of HIGH severity because funds are permanently locked

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.

arjun-io

Fixed https://github.com/equilibria-xyz/perennial-v2/pull/79

re: Escalation - since this contract can be upgraded the funds are not permanently locked

Emedudu

> Fixed https://github.com/equilibria-xyz/perennial-v2/pull/79

Can't access the repo to review the fix. It's probably a private repo.

> since this contract can be upgraded the funds are not permanently locked

While it's true that the contract can potentially be upgraded to address this issue, it's essential to acknowledge that the current code we audited does, in fact, contain a high severity vulnerability. Otherwise, implying that all upgradeable contracts are free of bugs simply because they can be upgraded to resolve them would be misleading.

arjun-io

> Otherwise, implying that all upgradeable contracts are free of bugs simply because they can be upgraded to resolve them would be misleading.

The distinction here is that "funds stuck" are fixable via upgrades, whereas attacks which immediately drain funds or those which cause accounting errors are not after they are executed.

hrishibhat

Result: High Has duplicates Although Sponsor raises a valid point, perhaps Sherlock needs to have a rule with respect to smart contract upgrade-related

issues. Historically smart contract upgrades have not weighed on the issue severity decisions but will be considered in future rule updates, however, this issue will be considered as a valid high issue based on historical decisions on similar issues.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- Emedudu: accepted

jacksanford1

From WatchPug:

Fixed

Issue H-4: PythOracle:if price.expo is less than 0, wrong prices will be recorded

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/56

Found by

Emmanuel, minhtrng, panprog In PythOracle#_recordPrice function, prices with negative exponents are not handled correctly, leading to a massive deviation in prices.

Vulnerability Detail

Here is PythOracle#_recordPrice function:

```
function _recordPrice(uint256 oracleVersion, PythStructs.Price memory price)
↪   private {
    _prices[oracleVersion] = Fixed6Lib.from(price.price).mul(
        Fixed6Lib.from(SafeCast.toInt256(10 ** SafeCast.toUint256(price.expo > 0
↪   ? price.expo : -price.expo)))
    );
    _publishTimes[oracleVersion] = price.publishTime;
}
```

If price is 5e-5 for example, it will be recorded as 5e5 If price is 5e-6, it will be recorded as 5e6.

As we can see, there is a massive deviation in recorded price from actual price whenever price's exponent is negative

Impact

Wrong prices will be recorded. For example, If priceA is 5e-5, and priceB is 5e-6. But due to the wrong conversion,

- There is a massive change in price(5e5 against 5e-5)
- we know that priceA is ten times larger than priceB, but priceA will be recorded as ten times smaller than priceB. Unfortunately, current payoff functions may not be able to take care of these discrepancies

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L203

Tool used

Manual Review

Recommendation

In PythOracle.sol, `_prices` mapping should not be `mapping(uint256 => Fixed6) private _prices;` Instead, it should be `mapping(uint256 => Price) private _prices;`, where Price is a struct that stores the price and expo:

```
struct Price{
    Fixed6 price,
    int256 expo
}
```

This way, the price exponents will be preserved, and can be used to scale the prices correctly wherever it is used.

Discussion

sherlock-admin

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

141345 commented:

> h

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/53

jacksanford1

From WatchPug:

The updated version may lower the precision from a higher value (e.g. 10) to the fixed precision of 6.

For example, the price of BTT/USD would be rounding down to 0 with the updated version.

https://pyth.network/price-feeds/crypto-btt-usd

arjun-io

> From WatchPug:
>
> The updated version may lower the precision from a higher value (e.g. 10) to the fixed precision of 6.

> For example, the price of BTT/USD would be rounding down to 0 with the updated version.
>
> https://pyth.network/price-feeds/crypto-btt-usd

Ack, noted that we currently do not support price feeds with very small unit prices. We'll record this among the limitations / requirements for new markets and add it to the list of possible future improvements, but will not implement a fix at this time.

Issue H-5: Vault.sol: `settle`ing the 0 address will disrupt accounting

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/62

Found by

Emmanuel, WATCHPUG, bin2chen, moneyversed Due to the ability of anyone to settle the 0 address, the global assets and global shares will be wrong because lower keeper fees were deducted within the `_settle` function.

Vulnerability Detail

Within `Vault#_loadContext` function, the context.global is the account of the 0 address, while context.local is the account of the address to be updated or settled:

```
function _loadContext(address account) private view returns (Context memory
↪   context) {
    ...
    context.global = _accounts[address(0)].read();
    context.local = _accounts[account].read();
    context.latestCheckpoint = _checkpoints[context.global.latest].read();
}
```

If a user settles the 0 address, the global account will be updated with wrong data.

Here is the _settle logic:

```
function _settle(Context memory context) private {
    // settle global positions
    while (
        context.global.current > context.global.latest &&
        _mappings[context.global.latest + 1].read().ready(context.latestIds)
    ) {
        uint256 newLatestId = context.global.latest + 1;
        context.latestCheckpoint = _checkpoints[newLatestId].read();
        (Fixed6 collateralAtId, UFixed6 feeAtId, UFixed6 keeperAtId) =
↪   _collateralAtId(context, newLatestId);
        context.latestCheckpoint.complete(collateralAtId, feeAtId, keeperAtId);
        context.global.processGlobal(
            newLatestId,
            context.latestCheckpoint,
            context.latestCheckpoint.deposit,
            context.latestCheckpoint.redemption
        );
        _checkpoints[newLatestId].store(context.latestCheckpoint);
```

```
    }

    // settle local position
    if (
        context.local.current > context.local.latest &&
        _mappings[context.local.current].read().ready(context.latestIds)
    ) {
        uint256 newLatestId = context.local.current;
        Checkpoint memory checkpoint = _checkpoints[newLatestId].read();
        context.local.processLocal(
            newLatestId,
            checkpoint,
            context.local.deposit,
            context.local.redemption
        );
    }
}
```

If settle is called on 0 address, _loadContext will give context.global and context.local same data. In the _settle logic, after the global account(0 address) is updated with the correct data in the `while` loop(specifically through the processGlobal function), the global account gets reupdated with wrong data within the `if` statement through the processLocal function.

Wrong assets and shares will be recorded. The global account's assets and shares should be calculated with toAssetsGlobal and toSharesGlobal respectively, but now, they are calculated with toAssetsLocal and toSharesLocal.

toAssetsGlobal subtracts the globalKeeperFees from the global deposited assets, while toAssetsLocal subtracts globalKeeperFees/Checkpoint.count fees from the local account's assets.

So in the case of settling the 0 address, where global account and local account are both 0 address, within the while loop of _settle function, depositedAssets-globalKeeperFees is recorded for address(0), but then, in the `if` statement, depositedAssets-(globalAssets/Checkpoint.count) is recorded for address(0)

And within the `Vault#_saveContext` function, context.global is saved before context.local, so in this case, context.global(which is 0 address with correct data) is overridden with context.local(which is 0 address with wrong data).

Impact

The global account will be updated with wrong data, that is, global assets and shares will be higher than it should be because lower keeper fees was deducted.

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L190 https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/Vault.sol#L315 https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/types/Checkpoint.sol#L99 https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-vault/contracts/types/Checkpoint.sol#L119

Tool used

Manual Review

Recommendation

I believe that the ability to settle the 0 address is intended, so an easy fix is to save local context before saving global context: Before:

```
function _saveContext(Context memory context, address account) private {
    _accounts[address(0)].store(context.global);
    _accounts[account].store(context.local);
    _checkpoints[context.currentId].store(context.currentCheckpoint);
}
```

After:

```
function _saveContext(Context memory context, address account) private {
    _accounts[account].store(context.local);
    _accounts[address(0)].store(context.global);
    _checkpoints[context.currentId].store(context.currentCheckpoint);
}
```

Discussion

sherlock-admin

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

141345 commented:

> h

arjun-io

Great find - we'll fix this

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/86

jacksanford1

From WatchPug:

Fixed.

Issue H-6: Keepers will suffer significant losses due to miss compensation for L1 rollup fees

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/91

Found by

KingNFT While keepers submits transactions to L2 EVM chains, they need to pay both L2 execution fee and L1 rollup fee. Actually, L1 fees are much higher than L2 fees. In many case, L2 fees can be practically negligible. The current implementation only compensate and incentive keepers based on L2 gas consumption, keepers will suffer significant losses.

Vulnerability Detail

As shown of `keep()` modifier (L40-57), only L2 execution fee are compensated.

```
File: perennial-v2\packages\perennial-oracle\contracts\pyth\PythOracle.sol
124:     function commitRequested(uint256 versionIndex, bytes calldata
↪  updateData)
125:         public
126:         payable
127:         keep(KEEPER_REWARD_PREMIUM, KEEPER_BUFFER, "")
128:     {
...
157:     }

File: perennial-v2\packages\perennial-extensions\contracts\MultiInvoker.sol
359:     function _executeOrder(
360:         address account,
361:         IMarket market,
362:         uint256 nonce
363:     ) internal keep (
364:         UFixed18Lib.from(keeperMultiplier),
365:         GAS_BUFFER,
366:         abi.encode(account, market, orders(account, market, nonce).fee)
367:     ) {
...
384:     }


File: root\contracts\attribute\Kept.sol
40:     modifier keep(UFixed18 multiplier, uint256 buffer, bytes memory data) {
41:         uint256 startGas = gasleft();
42:
43:         _;
```

```
44:
45:         uint256 gasUsed = startGas - gasleft();
46:         UFixed18 keeperFee = UFixed18Lib.from(gasUsed)
47:             .mul(multiplier)
48:             .add(UFixed18Lib.from(buffer))
49:             .mul(_etherPrice())
50:             .mul(UFixed18.wrap(block.basefee));
51:
52:         _raiseKeeperFee(keeperFee, data);
53:
54:         keeperToken().push(msg.sender, keeperFee);
55:
56:         emit KeeperCall(msg.sender, gasUsed, multiplier, buffer, keeperFee);
57:     }
```

Takes a random selected transaction at the writing time https://optimistic.etherscan.io/tx/0xbb8e68e21c92acf4171fb6041b758b55acc3c559ec4595ed1129d534d90de995

We can find the L1 fee is much expensive than L2 fee.

```
L2 fee = 0.06 Gwei * 113,449 ~= 6,806 Gwei
L1 fee = 0.000006565634612417 ETH ~= 65656 Gwei
L1 / L2 = 964%
```

Typically, L1 fees are dynamic and determined by the calldata length and smoothed Ethereum gas prices. To submit Pyth oracle price, the `VAA` calldata will be 1700+ bytes, keepers need to pay much L1 rollup fee.

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/test/integration/pyth/PythOracle.test.ts#L34

Impact

No enough incentive for keeper to submit oracle price and execute orders, the system will not work.

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/root/contracts/attribute/Kept.sol#L40

Tool used

Manual Review

Recommendation

Compensating L1 rollup fee, here are some reference: https://docs.arbitrum.io/arbos/l1-pricing https://community.optimism.io/docs/developers/build/transaction-fees/#the-l1-data-fee

Discussion

sherlock-admin

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

141345 commented:

> m

arjun-io

The `Kept` helper supports a buffer amount which can be used for this use case.

ydspa

Escalate

Using `Kept` buffer can't solve this problem, as the ratio of `L1GasPrice / L2GasPrice` changes in every block and can vary with a extreme wide range. Let's say

```
CurrentL1GasPrice = 20 Gwei // it refers to smoothed recent Ethereum gas price
↪  provided by L2 system contract
CurretnL2GasPrice = 2 Gwei   // it refers to block.basefee
L1RollupGas = 40,000
L1RollupFee = L1RollupGas * CurrentL1GasPrice  = 40,000 * 20 = 800,000 Gwei
```

To compensate L1 rollup fee, we should set

```
buffer = L1RollupFee / CurrentL2GasPrice = 800,000 / 2 = 400,000
```

to make

```
L1Compensation = buffer *  CurrentL2GasPrice = 400,000 * 2 = 800,000 =
↪  L1RollupFee
```

some time later, i.e. 1 hour, gas prices change to

```
NewL1GasPrice = 40 Gwei
NewL2GasPrice = 0.01 Gwei
```

we can see the `L1Compensation` will be far less than `L1RollupFee`, keepers suffer losses

```
L1RollupFee = L1RollupGas * CurrentL1GasPrice = 40, 000 * 40 = 1,600,000 Gwei
L1Compensation = buffer * CurretnL2GasPrice = 400, 000 * 0.01 = 4,000 Gwei
```

In contrast, if gas prices change to

```
NewL1GasPrice = 10 Gwei
NewL2GasPrice = 2 Gwei
```

then, the `L1Compensation` is larger than `L1RollupFee`, keepers earn extra profit.

```
L1RollupFee = L1RollupGas * CurrentL1GasPrice = 40,000 * 10 = 400,000 Gwei
L1Compensation= buffer * CurretnL2GasPrice = 400,000 * 2 = 800,000 Gwei
```

Therefore, regardless of how we set `buffer`, as it's a fixed value, sometimes keepers suffer losses, and other times keepers earn extra profit.

During periods that keepers suffer losses, they are likely to stop working and the system is blocked. And other periods , the protocol suffer losses.

About the severity: (1) As the users' orders will be held until keepers become profitable, the waiting time may be short as some minutes, may be long as some hours, or even some days. Users' financial losses are foreseeable. (2) while keepers earn extra profit, proper ratio of `L1GasPrice / L2GasPrice` and `block.basefee` could make `keeperReward > settlementFee`, then a new attack vector become feasiblekeepers can set `1 wei` orders by self to drain fund from protocol, as `self.fee ~= 0, profit ~= keeperReward - self.keeper = keeperReward - settlementFee`

```
File: perennial-v2\packages\perennial\contracts\types\Order.sol
50:     function registerFee(
51:         Order memory self,
52:         OracleVersion memory latestVersion,
53:         MarketParameter memory marketParameter,
54:         RiskParameter memory riskParameter
55:     ) internal pure {
...
64:         self.fee = self.maker.abs().mul(latestVersion.price.abs()).mul(UFixe
↪  d6Lib.from(makerFee))
65:             .add(self.long.abs().add(self.short.abs()).mul(latestVersion.pri
↪  ce.abs()).mul(UFixed6Lib.from(takerFee)));
66:
67:         self.keeper = isEmpty(self) ? UFixed6Lib.ZERO :
↪  marketParameter.settlementFee;
68:     }
```

To sum up, I think it's high, not medium

sherlock-admin2

> Escalate
>
> Using `Kept` buffer can't solve this problem, as the ratio of `L1GasPrice / L2GasPrice` changes in every block and can vary with a extreme wide range. Let's say
>
> ```
> CurrentL1GasPrice = 20 Gwei // it refers to smoothed recent Ethereum gas
> ↪  price provided by L2 system contract
> CurretnL2GasPrice = 2 Gwei   // it refers to block.basefee
> L1RollupGas = 40,000
> L1RollupFee = L1RollupGas * CurrentL1GasPrice  = 40,000 * 20 = 800,000 Gwei
> ```
>
> To compensate L1 rollup fee, we should set
>
> ```
> buffer = L1RollupFee / CurrentL2GasPrice = 800,000 / 2 = 400,000
> ```
>
> to make
>
> ```
> L1Compensation = buffer *  CurrentL2GasPrice = 400,000 * 2 = 800,000 =
> ↪  L1RollupFee
> ```
>
> some time later, i.e. 1 hour, gas prices change to
>
> ```
> NewL1GasPrice = 40 Gwei
> NewL2GasPrice = 0.01 Gwei
> ```
>
> we can see the `L1Compensation` will be far less than `L1RollupFee`, keepers suffer losses
>
> ```
> L1RollupFee = L1RollupGas * CurrentL1GasPrice = 40, 000 * 40 = 1,600,000
> ↪  Gwei
> L1Compensation = buffer * CurretnL2GasPrice = 400, 000 * 0.01 = 4,000 Gwei
> ```
>
> In contrast, if gas prices change to
>
> ```
> NewL1GasPrice = 10 Gwei
> NewL2GasPrice = 2 Gwei
> ```
>
> then, the `L1Compensation` is larger than `L1RollupFee`, keepers earn extra profit.
>
> ```
> L1RollupFee = L1RollupGas * CurrentL1GasPrice = 40,000 * 10 = 400,000 Gwei
> L1Compensation= buffer * CurretnL2GasPrice = 400,000 * 2 = 800,000 Gwei
> ```


Therefore, regardless of how we set `buffer`, as it's a fixed value, sometimes keepers suffer losses, and other times keepers earn extra profit.

During periods that keepers suffer losses, they are likely to stop working and the system is blocked. And other periods , the protocol suffer losses.

About the severity: (1) As the users' orders will be held until keepers become profitable, the waiting time may be short as some minutes, may be long as some hours, or even some days. Users' financial losses are foreseeable. (2) while keepers earn extra profit, proper ratio of `L1GasPrice / L2GasPrice` and `block.basefee` could make `keeperReward > settlementFee`, then a new attack vector become feasiblekeepers can set `1 wei` orders by self to drain fund from protocol, as `self.fee ~= 0`, `profit ~= keeperReward - self.keeper = keeperReward - settlementFee`

```
File: perennial-v2\packages\perennial\contracts\types\Order.sol
50:     function registerFee(
51:         Order memory self,
52:         OracleVersion memory latestVersion,
53:         MarketParameter memory marketParameter,
54:         RiskParameter memory riskParameter
55:     ) internal pure {
...
64:         self.fee = self.maker.abs().mul(latestVersion.price.abs()).mul(
↪  UFixed6Lib.from(makerFee))
65:             .add(self.long.abs().add(self.short.abs()).mul(latestVersio
↪  n.price.abs()).mul(UFixed6Lib.from(takerFee)));
66:
67:         self.keeper = isEmpty(self) ? UFixed6Lib.ZERO :
↪  marketParameter.settlementFee;
68:     }
```

To sum up, I think it's high, not medium

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.

arjun-io

Thanks for the thorough explanation on why a buffer won't work. We agree that we should modify the Kept function to estimate L1 gas costs, thanks for the report!

141345

The severity of medium seems more appropriate.

Because according to sherlock's HM criteria:

> Medium: viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future.
>
> High: This vulnerability would result in a material loss of funds, and the cost of the attack is low.

Here the imbalanced L1/L2 gas fee would not be too frequent. Most of the time, the buffer will be good to compensate for the keeper. The described loss will incur when all the special conditions are met, and the fixed buffer is not enough to cover the keeper's cost.

arjun-io

Fixed Update Kept for L2s: https://github.com/equilibria-xyz/root/pull/74 and https://github.com/equilibria-xyz/root/pull/76 Update Pyth Oracle for L2: https://github.com/equilibria-xyz/perennial-v2/pull/85

hrishibhat

@ydspa tend to agree with this comment: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/91#issuecomment-1697458799 A valid medium

ydspa

> @ydspa tend to agree with this comment: #91 (comment) A valid medium

I think the conclusion of `Here the imbalanced L1/L2 gas fee would not be too frequent` is not correct, the ratio of `L1GasPrice / L2GasPrice` vary frequently in a very wide range. The following table shows 3 typical `2-hours` range of `Ethereum VS Optimistic` gas price in the past two weeks. We can see the `Ratio` can vary from `e1` to as large as `e9`.

() DateTime	L1GasPrice	L2GasPrice(Base)	Ratio(L1/L2)	Link
()				
2023-08-27 02:00	13.19 Gwei	55 wei	2.40e8	link
2023-08-27 02:30	16.20 Gwei	56 wei	2.30e8	link
2023-08-27 03:00	14.95 Gwei	58 wei	2.58e8	link
2023-08-27 03:30	15.55 Gwei	53 wei	2.93e8	link
2023-08-27 04:00	21.53 Gwei	51 wei	4.06e8	link

() DateTime	L1GasPrice	L2GasPrice(Base)	Ratio(L1/L2)	Link
()				
2023-08-22 02:00	46.41 Gwei	66 wei	0.7e9	link
2023-08-22 02:30	121.65 Gwei	53 wei	2.30e9	link
2023-08-22 03:00	89.76 Gwei	52 wei	1.73e9	link
2023-08-22 03:30	68.45 Gwei	51 wei	1.34e9	link
2023-08-22 04:00	42.59 Gwei	100 wei	0.43e9	link
2023-08-17 10:00	229.22 Gwei	3.36 Gwei	6.82e1	link
2023-08-17 10:30	121.38 Gwei	0.30 Gwei	4.05e2	link
2023-08-17 11:00	138.89 Gwei	1.64 Gwei	8.47e1	link
2023-08-17 11:30	61.60 Gwei	0.1 Gwei	6.16e2	link
2023-08-17 12:00	36.84 Gwei	0.0058 Gwei	6.35e3	link

()

It will cause users' orders been held up to some hours in 2 ways: Let's say we set `buffer` according `Ratio = 2e8` and plus up to `100%` intended reward for keepers, which means the system can only work well while `Ratio` is during `(2e8, 4e8)`. So, in `2023-08-27 02:00 ~ 04:00` the system works, but in most cases, the system doesn't work such as (1) in `2023-08-22 02:00 ~ 04:00`, the `Ratio` is too high, keepers would suffer loss which might lead them to stop pushing price and users' orders are held. (2) in `2023-08-17 10:00 ~ 12:00`, the `Ratio` is too low, keepers earn too much reward, which will trigger the following protection, then no new price can be pushed to chain, all users' orders are held entirely.

```
File: packages\perennial-oracle\contracts\OracleFactory.sol
93:     function claim(UFixed6 amount) external {
94:         if (amount.gt(maxClaim)) revert OracleFactoryClaimTooLargeError();
...
97:     }
```

To be emphasized:

1. Regardless of how we set `buffer`, it can only work in a extreme narrow `Ratio` range as compared to `(e1, e9)`.
2. Just in the past 2 weeks, we can easily find more examples like the above ones that would block users' orders for up to 2 hours. It's almost `100%` sure this bug would be repeatedly triggered during the long lifetime of the protocol.
3. As a perpetual exchange APP, users' orders are held up to some hours, especially in the above `case (2)`, the service is stopped entirely, users' financial losses are foreseeable, the impact is high.

Hence, the issue is with both `high` probability of occurrence and `high` impact, undoubtedly should be a `high` issue.

141345

Yes the fee ratio could vary across a wide span. Extreme high L1 gas fee is intermittent, and will repeat. However, the protocol can set some fix buffer at 90% percentile of the L1/L2 fee ratio (maybe at e8 level), then the loss will only emerge in some certain scenarios.

Still seems medium due to conditional loss.

ydspa

> Yes the fee ratio could vary across a wide span. Extreme high L1 gas fee is intermittent, and will repeat. However, the protocol can set some fix buffer at 90% percentile of the L1/L2 fee ratio (maybe at e8 level), then the loss will only emerge in some certain scenarios.
>
> Still seems medium due to conditional loss.

(1I'm convinced that you can't find a `buffer` that works at 90% percentile, it is even hard to work at 50%, you can set buffer with a fixed Ratio plus some intended reward such as 100% (in perennial testcase, it's only 50%), then verify data in the past 1 year

(2) conditional loss of `1 %` probability VS `99%` probability are not the same thing, i think the later should be treated as `unconditional .`

141345

Even the distribution is Pareto distribution (power law), percentile still works. 50%, 90% can be found.

ydspa

> Even the distribution is Pareto distribution (power law), percentile still works. 50%, 90% can be found.

Give your specific parameters please

141345

> you can't find a `buffer` that works at 90% percentile, it is even hard to work at 50%

percentile always can be found, such as medium number(50%).

On the opposite, mean value could be hard to find, those extreme high gas fee can make mean not converge.

ydspa

More proof: if we set buffer according `e8` level as recommended above, then we can find lots of examples the actual `Ratio` falls below `e6` which keepers would earn too much rewards (10,000% level) to trigger system protection to block price submission.

The instances are all from the past one month, and sample interval is 1 hour, that's mean, averagely speaking, the system will be blocked for about 127 hours (5+ days) in one month if we set buffer according `e8` level.

```
index  blockHeight
0 108995426
1 108952226
2 108865826
3 108864026
4 108862226
5 108860426
6 108822626
7 108820826
8 108819026
9 108817226
10 108815426
11 108813626
12 108811826
13 108810026
14 108808226
15 108806426
16 108804626
17 108802826
18 108801026
19 108799226
20 108604826
21 108570626
22 108504026
```

```
23 108471626
24 108376226
25 108360026
26 108358226
27 108356426
28 108345626
29 108318626
30 108311426
31 108309626
32 108298826
33 108297026
34 108295226
35 108293426
36 108291626
37 108289826
38 108288026
39 108286226
40 108284426
41 108268226
42 108266426
43 108264626
44 108262826
45 108261026
46 108259226
47 108257426
48 108255626
49 108253826
50 108252026
51 108250226
52 108248426
53 108246626
54 108244826
55 108243026
56 108241226
57 108239426
58 108237626
59 108235826
60 108234026
61 108232226
62 108230426
63 108223226
64 108221426
65 108219626
66 108217826
67 108216026
68 108214226
69 108212426
70 108210626
```


71 108208826
72 108207026
73 108205226
74 108203426
75 108201626
76 108079226
77 107958626
78 107953226
79 107951426
80 107949626
81 107947826
82 107946026
83 107944226
84 107942426
85 107940626
86 107938826
87 107937026
88 107935226
89 107933426
90 107931626
91 107929826
92 107928026
93 107926226
94 107924426
95 107922626
96 107920826
97 107919026
98 107917226
99 107915426
100 107913626
101 107911826
102 107910026
103 107908226
104 107906426
105 107904626
106 107902826
107 107901026
108 107870426
109 107780426
110 107778626
111 107776826
112 107775026
113 107773226
114 107771426
115 107769626
116 107767826
117 107742626
118 107740826

```
119 107739026
120 107737226
121 107735426
122 107733626
123 107731826
124 107730026
125 107728226
126 107726426
```

141345

The main point is, there is some parameter, considering the trade off, can balance the loss and overpayment. e8 is just an example.

ydspa

> The main point is, there is some parameter, considering the trade off, can balance the loss and overpayment. e8 is just an example.

What my meaning is the parameter doesn't exist at all, mathematically speaking, `L1Price` and `L2Price` are `enough independent`, we can't calculate the following result by only `L2Price`

```
Optimism Transaction Fee = [ Fee Scalar * L1 Gas Price * (Calldata + Fixed
↪  Overhead) ] + [ L2 Gas Price * L2 Gas Used ]
```

reference: https://dune.com/haddis3/optimism-fee-calculator

This is why no matter we set `buffer` according `e1`, `e2`, ..., `e9` or any other value, the issue would be repeatedly triggered. So, actually it's a `unconditional` issue.

ydspa

> > you can't find a `buffer` that works at 90% percentile, it is even hard to work at 50%
>
> percentile always can be found, such as medium number(50%).
>
> On the opposite, mean value could be hard to find, those extreme high gas fee can make mean not converge.

Here is a mistake in thoughts, set `buffer` according the number of 90% percentile, not means the setting will work from 0% percentile to 90% percentile, actually it might only work for `(85%, 95%)`.

141345

0.001, 0.02, 0.3, 4, 5, 6, 7, 8, 9, 10000.

For this skewed distribution, 90% percentile is 9. Why only work for around 9?

ydspa

> 0.001, 0.02, 0.3, 4, 5, 6, 7, 8, 9, 10000.
>
> For this skewed distribution, 90% percentile is 9. Why only work for around 9? Let's say, at `e9`, keeper get $1, then at `e8` it becomes $10, at `e7` it would be $100...

141345

> repeatedly triggered

My understanding, something like, it could be triggered 10-20% of the time in a certain time span, such as 1 month, 1 week.

It still falls into the category with condition: Per criteria

> The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future.

ydspa

My understanding about the probability of issue occurrence, which i learned when i work for other audit firms, are like this:

If the issue will occur even `once` with `high probability` in a reasonable period such as 3 months, then it's high probability.

It's far away from the thought that only if a issue will keep occurring in most the time, let's say, in 2 of the 3 months, then it's high probability.

So, in my opinion, if a issue might occur hundred times a month, it's obviously a `high probability`. And with `high probability` and `high impact` issue, we mark it as `high`.

Actually, we can think the situation much more simple:

```
If an exchange's service would be suspended randomly sum up to average 7 days per month, is it critical, high or medium?
```

I will choose critical though Sherlock doesn't have this level.

hrishibhat

Result: High Unique After considering further points raised by Watson given the frequency of the condition mentioned in the example above. The impact is not just about the loss for the keeper but also keepers not submitting txs and executing orders. This can be looked at as a severe issue overall. Considering this issue a valid high.

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- ydspa: accepted

jacksanford1

From WatchPug:

https://github.com/equilibria-xyz/root/blob/ea30036560e54a9eb951a77c7a104e5cfe3c4d72/contracts/attribute/Kept/Kept.sol#L44-L62

https://github.com/equilibria-xyz/perennial-v2/blob/602218ca8cb62db649bf4b6a6722824e9ab20166/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L129-L133

The dynamicCalldata passed to the keep modifier is not the entire calldata. This will result in a wrong result of _calculateDynamicFee().

Recommendation Consider calculating the length of the calldata before calling keep and change bytes memory dynamicCalldata to uint256 calldataLength.

arjun-io

> From WatchPug:
>
> https://github.com/equilibria-xyz/root/blob/ea30036560e54a9eb951a77c7a104e5cfe3c4d72/contracts/attribute/Kept/Kept.sol#L44-L62
>
> **Recommendation** Consider calculating the length of the calldata before calling keep and change bytes memory dynamicCalldata to uint256 calldataLength.

Passing in the whole msg.data is vulnerable to data expansion attacks, we chose to instead let the user of the library choose which portion of the data to incentivize. This is similar to how only the function with the incentivize modifier is counted with respect to the gas usage instead of the entire transaction.

The difference in real cost to the keeper can be accounted for via the premium and buffer parameters.

Issue M-1: `_unwrap` in `MultiInvoker.sol` can revert every time in some cases which will make the users not being able to `_liquidate` or `_withdraw` with `warp` to true

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/22

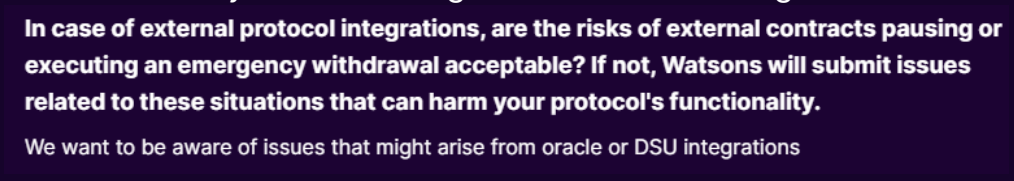
Found by

Vagner `_withdraw` makes some wrong assumptions which could lead to reverts all the time if `DSU` protocol will be in debt.

Vulnerability Detail

The function `_withdraw` is called in `_liquidate`, `_vaultUpdate` and `_update`, and if the `wrap` is set to true, which will be all the time in the case of `_liquidate`, it will try to call `_unwrap` https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L262 After that `_unwrap` will check if the address of `batcher` is 0, which can be the case if it is not set up in the constructor, or if the `balanceOf` USDC of `batcher` is less than the amount intended to withdraw https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L287 and if any of that will be true it will try to call the `redeem` function on `reserve` with the intended amount https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L288 and then transfers the `amount` to the `receiver` https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L289 The problem relies in the fact that the protocol assumes that the `amount` that is used in `redeem` function will always be equal to the `amount` of USDC returned, which is not the case. As can be seen in the code of the `reserve` here the `redeemAmount` is a calculation that depends on the `redeemPrice` which depends on the debt of the protocol https://github.com/emptysetsquad/emptyset/blob/c5d876fbd8ff1fac988898b77ef5461971f9fdd2/protocol/contracts/src/reserve/ReserveComptroller.sol#L125 The whole calculation of `redeemPrice` is done depending of the `reserveRatio` of the Protocol https://github.com/emptysetsquad/emptyset/blob/c5d876fbd8ff1fac988898b77ef5461971f9fdd2/protocol/contracts/src/reserve/ReserveComptroller.sol#L81-L84 so in the case where DSU will make some bad decisions and it will be in debt the `redeemAmount` calculated will be less than the actual amount inserted into `redeem` function, and that `redeemAmount` will be the one actual transferred to the `MultiInvoker.sol` https://github.com/emptysetsquad/emptyset/blob/c5d876fbd8ff1fac988898b77ef5461971f9fdd2/protocol/contracts/src/reserve/ReserveComptroller.sol#L130 So when `_unwrap` will try to transfers the same amount of USDC https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L289 the transaction would

revert because the contract would not have enough USDC in the contract, making `_vaultUpdate`, `_update` with `warp` to true on `_withdraw` reverting all the time and `_liquidate` reverting 100% of the time. Since the protocol stated that they want to be aware of any issue that might arise with the integration of DSU



I decided to specify this issue which could happen can cause a lot of damage to the users.

Impact

Impact is a high one since it will cause the liquidation process to fail and also withdrawing funds to fail.

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-extensions/contracts/MultiInvoker.sol#L285-L289

Tool used

Manual Review

Recommendation

Since the contract doesn't store any USDC, you could transfer the whole balance of the contract every time after you call `redeem` or do balance before and balance after, and transfer the difference. It is important to take special care when you implement other protocols since any failures will also break your protocol.

Discussion

sherlock-admin

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

141345 commented:

> I

n33k commented:

> low

panprog commented:

> medium because the situation is very unlikely

VagnerAndrei26

Escalate : I believe this issue to be valid and at least a valid medium under the considerations of Sherlock rules which states

> **Medium:** There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

since the external protocol the Perennial team is working, DSU, is not controlled by the them and since they specified in the ReadMe that they want to know any issues that might arise because of DSU integration,

> **In case of external protocol integrations, are the risks of external contracts pausing executing an emergency withdrawal acceptable? If not, Watsons will submit issues related to these situations that can harm your protocol's functionality.**
>
> We want to be aware of issues that might arise from oracle or DSU integrations

which I provided in the report. The way the code is written right now might arise problems in case where DSU fails/ becomes insolvent and it can be solved pretty easily in the solution I provided, so considering all of this, this issue should be evaluated at least as a Medium since the impact is a high one, by blocking the whole liquidation process and withdrawing with wrap, and probability of happening can be low.

sherlock-admin2

> Escalate : I believe this issue to be valid and at least a valid medium under the considerations of Sherlock rules which states
>
> > **Medium:** There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.
>
> since the external protocol the Perennial team is working, DSU, is not controlled by the them and since they specified in the ReadMe that they want to know any issues that might arise because of DSU integration,
>
> > **In case of external protocol integrations, are the risks of external contracts pausing executing an emergency withdrawal acceptable? If not, Watsons will submit issues related to these situations that can harm your protocol's functionality.**
> >
> > We want to be aware of issues that might arise from oracle or DSU integrations
>
> which I provided in the report. The way the code is written right now might arise problems in case where DSU fails/ becomes insolvent and it can be solved pretty easily in the solution I provided, so considering all of this, this issue should be evaluated at least as a Medium since the impact is a high one, by blocking the whole liquidation process and withdrawing with wrap, and probability of happening can be low.

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.

141345

The condition is

> if it is not set up in the constructor, or if the balanceOf USDC of batcher is less than the amount intended to withdraw

But at first place, USDC is wrapped into DSU by batcher.

```
File: perennial-v2\packages\perennial-extensions\contracts\MultiInvoker.sol
271:     function _wrap(address receiver, UFixed18 amount) internal {

277:             // Wrap the USDC into DSU and return to the receiver
278:             batcher.wrap(amount, receiver);
```

So the USDC in batcher is expected to be fully backed.

Hence low/info severity is more appropriate.

VagnerAndrei26

> The condition is
>
> > if it is not set up in the constructor, or if the balanceOf USDC of batcher is less than the amount intended to withdraw
>
> But at first place, USDC is wrapped into DSU by batcher.
>
> ```
> File: perennial-v2\packages\perennial-extensions\contracts\MultiInvoker.sol
> 271:     function _wrap(address receiver, UFixed18 amount) internal {
>
> 277:             // Wrap the USDC into DSU and return to the receiver
> 278:             batcher.wrap(amount, receiver);
> ```
>
> So the USDC in batcher is expected to be fully backed.
>
> Hence low/info severity is more appropriate.

Yes that is true, but that is not the issue that I'm talking about here, I'm not saying that there will not be enough funds in the batcher to be paid back. The issue that I've talked here is the fact that DSU protocol can borrow DSU, which will increase the debt of the protocol, as can be seen here https://github.com/emptysetsquad/emptyset/blob/c5d876fbd8ff1fac988898b77ef5461971f9fdd2/protocol/contracts/src/reserve/ReserveComptroller.sol#L143-L150 in that case, when the `redeem` will be called and the redeem amount is calculated

`redeemPrice` which is used in the calculation can be less than one https://github.com/emptysetsquad/emptyset/blob/c5d876fbd8ff1fac988898b77ef5461971f9fdd2/protocol/contracts/src/reserve/ReserveComptroller.sol#L93-L95 , since the DSU protocol is in debt and redeeming will not be done 1-1, less USDC will be transferred to the `MultiInvoker.sol` than the `amount` specified in redeem. Because of that trying to push exactly the specific `amount` used in redeem, will fail and revert, because the contract will not have enough funds. That's why I specified that transferring the whole `balanceOf(address(this))` would solve this issue of blocking liquidation and withdrawing, since the contract doesn't hold any funds and just act as a middle man. So this is a risk associated with the DSU integration that can mess important functionalities of Perennial and also block funds.

141345

Sorry I misunderstood the report.

The DSU reserve has borrow function result in debt, and the redeem amount will not be enough. Or we can say it is when DSU depeg from USDC.

Seems an edge case issue.

VagnerAndrei26

> Sorry I misunderstood the report.
>
> The DSU reserve has borrow function result in debt, and the redeem amount will not be enough. Or we can say it is when DSU depeg from USDC.
>
> Seems an edge case issue.

Yep, it is an edge case, that's why in my escalation I specify that should be treated at least as a medium, since the damage exist and can be quite high and as for probability, it depends a lot on how DSU maintain their balance sheets, since the function can be called anytime by the owners, and since it was specify that Perennial team wants to know integration issue that could occur with DSU, it is in their advantage to protect their protocol against this case.

hrishibhat

@arjun-io

hrishibhat

Result: Medium Unique Considering this issue a valid medium given the edge case

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- VagnerAndrei26: accepted

jacksanford1

From WatchPug:

Acknowledged

Issue M-2: PythOracle `commit()` function doesn't require (nor stores) pyth price publish timestamp to be after the previous commit's publish timestamp, which makes it possible to manipulate price to unfairly liquidate users and possible stealing protocol funds

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/44

Found by

panprog

PythOracle allows any user to commit non-requested oracle version. However, it doesn't verify pyth price publish timestamp to be in order (like `commitRequested` does). This makes it possible to commit prices out of order, potentially leading to price manipulations allowing to unfairly liquidate users or steal funds from the protocol.

Vulnerability Detail

PythOracle `commitRequested()` has the following check: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L138-L139

However, `commit()` doesn't have the same check. This allows malicious user to commit prices out of order, which can potentially lead to price manipulation attacks and unfair liquidation of users or loss of protocol funds.

For example, the following scenario is possible: Timestamp = 100: Alice requests to open 1 ETH long position with $10 collateral Timestamp = 113: pyth price = $980 Timestamp = 114: pyth price = $990 Timestamp = 115: pyth price = $1000 Timestamp = 116: Keeper commits requested price = $1000 (with publish timestamp = 115) Timestamp = 117: Malicious user Bob commits oracle version 101 with price = $980 (publish timestamp = 113) and immediately liquidates Alice.

Even though the current price is $1000, Alice is liquidated using the price which is *earlier* than the price when Alice position is opened, which is unfair liquidation. The other more complex scenarios are also possible for malicious Bob to liquidate itself to steal protocol funds.

Impact

Unfair liquidation as described in the scenario above or possible loss of protocol funds.

Code Snippet

Tool used

Manual Review

Recommendation

Add publish time check and store publish time in `PythOracle.commit()`:

```
    function commit(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 > nextVersionIndexToCommit && oracleVersion >=
↪   versionList[nextVersionIndexToCommit]) {
            commitRequested(nextVersionIndexToCommit, updateData);
            return;
        }

+        if (pythPrice.publishTime <= _lastCommittedPublishTime) revert
↪   PythOracleNonIncreasingPublishTimes();
+        _lastCommittedPublishTime = pythPrice.publishTime;

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

Discussion

sherlock-admin

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

141345 commented:

> m

Emedudu

Escalate

This is an invalid issue.

> Malicious user Bob commits oracle version 101 with price = $980 (publish timestamp = 113) and immediately liquidates Alice.

How is Bob malicious? He committed the more recent price.

> Timestamp = 116: Keeper commits requested price = $1000 (with publish timestamp = 115)

If this price is used(which was requested at timestamp 100, when current timestamp is 115), it means that protocol is using a 15 seconds stale price. It is better(and an expected behaviour) when Bob commits a more recent price(price at timestamp 101) because now, Protocol is using a less stale price(14 seconds stale). So it is fair to liquidate Alice because even though the most recent price does not favour her, that's the rules.

sherlock-admin2

> Escalate
>
> This is an invalid issue.
>
> > Malicious user Bob commits oracle version 101 with price = $980 (publish timestamp = 113) and immediately liquidates Alice.
>
> How is Bob malicious? He committed the more recent price.
>
> > Timestamp = 116: Keeper commits requested price = $1000 (with publish timestamp = 115)
>
> If this price is used(which was requested at timestamp 100, when current timestamp is 115), it means that protocol is using a 15 seconds stale price. It is better(and an expected behaviour) when Bob commits a more recent price(price at timestamp 101) because now, Protocol is using a less stale price(14 seconds stale). So it is fair to liquidate Alice because even though the most recent price does not favour her, that's the rules.

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.

panprog

Escalate

This is a valid issue. Let me go into example in more details to explain. A few terms to better understand process:

- `oracle version` - the timestamp keeper commits the price to. It doesn't mean the price *at this* timestamp. It means the price used to settle positions at this timestamp.
- `pyth publish timestamp` - the timestamp of a price published and signed in the pyth network (actual time the price is observed). `publish timestamp` must come 12-15 seconds after `oracle version` (these are example settings in perennial tests).

Timestamp = 100: Alice requests to open 1 ETH long position with $10 collateral. Oracle request is created (with `oracle version` = 100, meaning only `publish prices` in 112..115 range can be commited) Timestamp = 113: `pyth publish price` = $980 Timestamp = 114: `pyth publish price` = $990 Timestamp = 115: `pyth publish price` = $1000 Timestamp = 116: Keeper commits requested price = $1000 (`oracle version` = 100, `publish timestamp` = 115) Timestamp = 117: Malicious user Bob commits (unrequested) `oracle version` = 101 with price = $980, `publish timestamp` = 113 and immediately liquidates Alice.

What happens is: `oracle version` = 100 has `publish timestamp` = 115 `oracle version` = 101 has `publish timestamp` = 113

This breaks invariant that publish timestamps must increase when oracle version increases.

So first position is opened using price $1000 (with `publish timestamp` = 115), then it is liquidated using the `publish timestamp = 113`, which is an earlier price than the price of when position was opened, which is unfair liquidation.

sherlock-admin2

> Escalate
>
> This is a valid issue. Let me go into example in more details to explain. A few terms to better understand process:
>
> - `oracle version` - the timestamp keeper commits the price to. It doesn't mean the price *at this* timestamp. It means the price used to settle positions at this timestamp.
> - `pyth publish timestamp` - the timestamp of a price published and signed in the pyth network (actual time the price is observed). `publish timestamp` must come 12-15 seconds after `oracle version` (these are example settings in perennial tests).
>
> Timestamp = 100: Alice requests to open 1 ETH long position with $10 collateral. Oracle request is created (with `oracle version` = 100, meaning only `publish prices` in 112..115 range can be commited) Timestamp = 113: `pyth publish price` = $980 Timestamp = 114: `pyth publish price` = $990 Timestamp = 115: `pyth publish price` = $1000 Timestamp = 116: Keeper commits requested price = $1000 (`oracle version` = 100, `publish timestamp` = 115) Timestamp = 117: Malicious user Bob commits (unrequested) `oracle version` = 101 with price = $980, `publish timestamp` = 113 and immediately liquidates Alice.
>
> What happens is: `oracle version` = 100 has `publish timestamp` = 115 `oracle version` = 101 has `publish timestamp` = 113
>
> This breaks invariant that publish timestamps must increase when oracle version increases.

> So first position is opened using price $1000 (with `publish timestamp = 115`), then it is liquidated using the `publish timestamp = 113`, which is an earlier price than the price of when position was opened, which is unfair liquidation.

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.

Emedudu

This assumption is predicated on the idea that there can be a rapid and significant change in the ETH/USD price, such as a $10 change as demonstrated in the example. Can we substantiate this assumption with historical data, demonstrating that the price of ETH can indeed fluctuate by $10 within a second, or even within 3 seconds?

Besides, it's crucial for the protocol to utilize the price from the most recent oracle version. This underscores the importance for users to maintain healthy positions. Referring to the example Watson provided, Alice should not let her position teeter on the brink of liquidation.

So again, Bob is not acting maliciously. He is diligently fulfilling his role as a keeper, which involves updating oracle versions. He happened upon a careless trader, Alice, with an unhealthy position and executed a liquidation. This action benefits the protocol by reducing the number of unhealthy positions.

Minh-Trng

While it is true that `commitRequested` enforces publish time to increase monotonically, while `commit` doesnt, the example shows how Alice has at one point in those 3 seconds been be below her liquidation point, so its absolutely fine (and beneficial for the protocol) for her to be liquidated.

Bob has done a better job at playing the keeper than the other keeper

panprog

> While it is true that `commitRequested` enforces publish time to increase monotonically, while `commit` doesnt, the example shows how Alice has at one point in those 3 seconds been be below her liquidation point, so its absolutely fine (and beneficial for the protocol) for her to be liquidated.

No, this is not true. If the price of $980 was commited first, then Alice position would be opened at a price of $980 and will not be liquidatable. If Alice was opened at a price of $990, then she also won't be liquidatable.

> This assumption is predicated on the idea that there can be a rapid and significant change in the ETH/USD price, such as a $10 change as

> demonstrated in the example. Can we substantiate this assumption with historical data, demonstrating that the price of ETH can indeed fluctuate by $10 within a second, or even within 3 seconds?

1. Even 0.01% of a price change is enough to liquidate the account. Yes, the user may be careless with leverage, however this doesn't make this liquidation fair. I chose more extreme numbers just to better demonstrate the scenario.
2. 3 seconds is just an example, it's a protocol setting and can be higher
3. Here is an extreme example to better understand why such liquidation is unfair. If the publish timestamp window is, say, 200 seconds long and the price starts dropping sharply from $1000 to $980. User expects the price to keep dropping, so he opens a short position at close to max leverage (so that a price move of $20 will get it liquidated). The price keeps falling sharply to $960. The position is opened at a price of $960 (end of 200 seconds interval) and immediately liquidated at a price of $980 (start of 200 seconds interval). I don't think any reasonable user expects his position to be liquidated using prices which were well before the opening of the position.

So even with 3-seconds time interval and much smaller price change magnitude, the same scenario is still possible and is still unfair. The published (observed) prices must come in the same order they're observed from pyth, they can not go in a random order.

Emedudu

> Even 0.01% of a price change is enough to liquidate the account. Yes, the user may be careless with leverage, however this doesn't make this liquidation fair. I chose more extreme numbers just to better demonstrate the scenario.

Why should a user leave his position to be liquidatable when there is a price change of 0.01%? This shows that the position is indeed unhealthy. One thing about oracles is that they return approximate values, and have little deviation from other oracles' values. So no reasonable trader will make her position liquidatable on a 0.01% price change.

> 3 seconds is just an example, it's a protocol setting and can be higher

From what I can see, they are CONSTANTS: MIN_VALID_TIME_AFTER_VERSION and MAX_VALID_TIME_AFTER_VERSION

> Here is an extreme example to better understand why such liquidation is unfair. If the publish timestamp window is, say, 200 seconds long and the price starts dropping sharply from $1000 to $980. User expects the price to keep dropping, so he opens a short position at close to max leverage (so that a price move of $20 will get it liquidated). The price keeps falling sharply to $960. The position is opened at a price of $960 (end of 200 seconds interval) and immediately liquidated at a price of

> $980 (start of 200 seconds interval). I don't think any reasonable user expects his position to be liquidated using prices which were well before the opening of the position

This seems unrealistic

Minh-Trng

A core assumption of this issue is that a user opens a position close to the liquidation price and/or in a highly volatile period without enough buffer till liquidation, so that even a price deviation within 2 seconds would liquidate them. this is clearly a user mistake rather than a protocol error. And thats why a medium severity is imo not justified.

A user does not know the oracle prices 12-15 seconds into the future, so they would not choose which price to be filled at and be mindful of having enough margin. The example might as well go like this:

Timestamp = 113: pyth publish price = $980 Timestamp = 114: pyth publish price = $990 Timestamp = 115: pyth publish price = $1000 Timestamp = 116: pyth publish price = $990 Timestamp = 117: pyth publish price = $980

In this case Alice might get filled at t=115 and get liquidated at t=117 and there is no one to blame but her.

To sum up, the issue is hypothetically possible but requires a careless user AND enough volatility within 2 seconds time span

panprog

> To sum up, the issue is hypothetically possible but requires a careless user AND enough volatility within 2 seconds time span

Yes, it's unlikely but possible, thus should be a valid medium.

Emedudu

Severity should be LOW because likelihood is low and impact is low

panprog

> Severity should be LOW because likelihood is low and impact is low

Impact is unfair liquidation, so it's high. For example, if the price starts dropping quickly -> every second the price will be less than or equal to previous second's price, in such situation opening short position at max leverage can be valid strategy for the user expecting continuaton of the price fall, and being liquidated at earlier price is also extremely unfair.

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/80

141345

Medium seems appropriate.

As per the discussion, it is plausible to commit prices out of order and cause loss, but with several conditions:

- limited time span
- high volatility
- the position is already on the border of liquidation

hrishibhat

Result: Medium Unique Considering this issue a valid medium based on the above discussion and the comment from Lead Judge

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- panprog: accepted
- Emedudu: rejected

jacksanford1

From WatchPug:

Fixed

Issue M-3: During oracle provider switch, if it is impossible to commit the last request of previous provider, then the oracle will get stuck (no price updates) without any possibility to fix it

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/46

Found by

panprog

When the oracle provider is updated (switched to 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 it's impossible to commit the latest request: for example, if pyth signature server is down for the period it is needed, or if all keepers were down for that time period, so valid price with signature for the timestamp required 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). It is also impossible to cancel provider switch as there is no such function. As such, the oracle price will get stuck and will never update, breaking the whole protocol with user funds stuck in the protocol.

Vulnerability Detail

The way oracle provider switch works is the following:

1. `Oracle.update()` is called to set a new provider. This is only allowed if there is no other provider switch pending.
2. There is a brief transition period, when both the previous provider and a new provider are active. This is to ensure that all the requests made to the previous oracle are commited before switching to a new provider. This is handled by the `Oracle._handleLatest()` function, in particular the switch to a new provider occurs only when `Oracle.latestStale()` returns true. The lines of interest to us are:

```
uint256 latestTimestamp = global.latest == 0 ? 0 :
↪   oracles[global.latest].provider.latest().timestamp;
if (uint256(oracles[global.latest].timestamp) > latestTimestamp) return false;
```

`latestTimestamp` - is the timestamp of last commited price for the previous provider `oracles[global.latest].timestamp` is the timestamp of the last requested price for the previous provider The switch doesn't occur, until last commited price is equal to or after the last request timestamp for the previous provider. 3. The functions to

commit the price are in PythOracle: `commitRequested` and `commit`. 3.1. `commitRequested` requires publish timestamp of the pyth price to be within `MIN_VALID_TIME_AFTER_VERSION..MAX_VALID_TIME_AFTER_VERSION` from *request time*. It is possible that pyth price with signature in this time period is not available for different reasons (pyth price feed is down, keeper was down during this period and didn't collect price and signature):

```
uint256 versionToCommit = versionList[versionIndex];
PythStructs.Price memory pythPrice = _validateAndGetPrice(versionToCommit,
↪  updateData);
```

`versionList` is an array of oracle request timestamps. And `_validateAndGetPrice()` filters the price within the interval specified (if it is not in the interval, it will revert):

```
return pyth.parsePriceFeedUpdates{value: pyth.getUpdateFee(updateDataList)}(
    updateDataList,
    idList,
    SafeCast.toUint64(oracleVersion + MIN_VALID_TIME_AFTER_VERSION),
    SafeCast.toUint64(oracleVersion + MAX_VALID_TIME_AFTER_VERSION)
)[0].price;
```

3.2. `commit` can not be done with timestamp older than the first oracle request timestamp: if any oracle request is still active, it will simply redirect to `commitRequested`:

```
if (versionList.length > nextVersionIndexToCommit && oracleVersion >=
↪  versionList[nextVersionIndexToCommit]) {
    commitRequested(nextVersionIndexToCommit, updateData);
    return;
}
```

4. All new oracle requests are directed to a **new** provider, this means that previous provider can not receive any new requests (which allows to finalize it):

```
function request(address account) external onlyAuthorized {
    (OracleVersion memory latestVersion, uint256 currentTimestamp) =
↪  oracles[global.current].provider.status();

    oracles[global.current].provider.request(account);
    oracles[global.current].timestamp = uint96(currentTimestamp);
    _updateLatest(latestVersion);
}
```

So the following scenario is possible: timestamp=69: oracle price is commited for timestamp=50 timestamp=70: user requests to open position (`Oracle.request()` is

made) timestamp=80: owner calls `Oracle.update()` timestamp=81: pyth price signing service goes offline (or keeper goes offline) ... timestamp=120: signing service goes online again. timestamp=121: another user requests to open position (`Oracle.request()` is made, directed to new provider) timestamp=200: new provider's price is commited (`commitRequested` is called with timestamp=121)

At this time, `Oracle.latest()` will return price at timestamp=50. It will ignore new provider's latest commit, because previous provider last request (timestamp=70) is still not commited. Any new price requests and commits to a new provider will be ignored, but the previous provider can not be commited due to absence of prices in the valid time range. It is also not possible to change oracle for the market, because there is no such function. It is also impossible to cancel provider update and impossible to change the provider back to previous one, as all of these will revert.

It is still possible for the owner to manually whitelist some address to call `request()` for the previous provider. However, this situation provides even worse result. While the latest version for the previous provider will now be later than the last request, so it will let the oracle switch to new provider, however `oracle.status()` will briefly return invalid oracle version, because it will return oracle version at the timestamp = last request before the provider switch, which will be invalid (the new request will be after that timestamp):

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/Oracle.sol#L103

This can be abused by some user who can backrun the previous provider oracle commit (or commit himself) and use the invalid oracle returned by `status()` (oracle version with price = 0). Market doesn't expect the oracle status to return invalid price (it is expected to be always valid), so it will use this invalid price as if it's a normal price = 0, which will totally break the market:

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L574-L577

So if the oracle provider switch becomes stuck, there is no way out and the market will become stale, not allowing any user to withdraw the funds.

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

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/Oracle.sol#L112-L113

Tool used

Manual Review

Recommendation

There are multiple possible ways to fix this. For example, allow to finalize previous provider if the latest commit from the new provider is newer than the latest commit from the previous provider by `GRACE_PERIOD` seconds. Or allow PythOracle to `commit` directly (instead of via `commitRequested`) if the commit oracleVersion is newer than the last request by `GRACE_PERIOD` seconds.

Discussion

sherlock-admin

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

141345 commented:

> m

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/58

jacksanford1

From WatchPug:

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.

Recommendation Consider adding a method on Oracle to cancel a requested version from the previous OracleProvider.

arjun-io

> From WatchPug:
>
> 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.
>
> **Recommendation** Consider adding a method on Oracle to cancel a requested version from the previous OracleProvider.

Similar to #145, 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.

Issue M-4: Bad debt (shortfall) liquidation leaves liquidated user in a negative collateral balance which can cause bank run and loss of funds for the last users to withdraw

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/72

Found by

WATCHPUG, panprog

Bad debt liquidation leaves liquidated user with a negative collateral. However, there is absolutely no incentive for anyone to repay this bad debt. This means that most of the time the account with negative balance will simply be abandoned. This means that this negative balance (bad debt) is taken from the other users, however it is not socialized, meaning that the first users to withdraw will be able to do so, but the last users will be unable to withdraw because protocol won't have enough funds. This means that any large bad debt in the market can trigger a bank run with the last users to withdraw losing their funds.

Vulnerability Detail

Consider the following scenario:

1. User1 and User2 are the only makers in the market each with maker=50 position and each with collateral=500. (price=$100)
2. A new user comes into the market and opens long=10 position with collateral=10.
3. Price drops to $90. Some liquidator liquidates the user, taking $10 liquidation fee. User is now left with the negative collateral = -$100
4. Since User1 and User2 were the other party for the user, each of them has a profit of $50 (both users have collateral=550)
5. At this point protocol has total funds from deposit of User1($500) + User2($500) + new user($10) - liquidator($10) = $1000. However, User1 and User2 have total collateral of 1100.
6. User1 closes position and withdraws $550. This succeeds. Protocol now has only $450 funds remaining and 550 collateral owed to User2.
7. User2 closes position and tries to withdraw $550, but fails, because protocol doesn't have enough funds. User2 can only withdraw $450, effectively losing $100.

Since all users know about this feature, after bad debt they will race to be the first to withdraw, triggering a bank run.

Impact

After **ANY** bad debt, the protocol collateral for all non-negative users will be higher than protocol funds available, which can cause a bank run and a loss of funds for the users who are the last to withdraw.

Even if someone covers the shortfall for the user with negative collateral, this doesn't guarantee absence of bank run:

1. If the shortfall is not covered quickly for any reason, the other users can notice disparency between collateral and funds in the protocol and start to withdraw
2. It is possible that bad debt is so high that any entity ("insurance fund") just won't have enough funds to cover it.

Proof of concept

The scenario above is demonstrated in the test, add this to test/unit/market/Market.test.ts:

```
it('panprog bad debt liquidation bankrun', async () => {

    function setupOracle(price: string, timestamp : number, nextTimestamp :
↪   number) {
        const oracleVersion = {
        price: parse6decimal(price),
        timestamp: timestamp,
        valid: true,
        }
        oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
        oracle.status.returns([oracleVersion, nextTimestamp])
        oracle.request.returns()
    }

    var riskParameter = {
        maintenance: parse6decimal('0.01'),
        takerFee: parse6decimal('0.00'),
        takerSkewFee: 0,
        takerImpactFee: 0,
        makerFee: parse6decimal('0.00'),
        makerImpactFee: 0,
        makerLimit: parse6decimal('1000'),
        efficiencyLimit: parse6decimal('0.2'),
        liquidationFee: parse6decimal('0.50'),
        minLiquidationFee: parse6decimal('10'),
        maxLiquidationFee: parse6decimal('1000'),
        utilizationCurve: {
        minRate: parse6decimal('0.0'),
        maxRate: parse6decimal('1.00'),
```

```
        targetRate: parse6decimal('0.10'),
        targetUtilization: parse6decimal('0.50'),
        },
        pController: {
        k: parse6decimal('40000'),
        max: parse6decimal('1.20'),
        },
        minMaintenance: parse6decimal('10'),
        virtualTaker: parse6decimal('0'),
        staleAfter: 14400,
        makerReceiveOnly: false,
    }
    var marketParameter = {
        fundingFee: parse6decimal('0.0'),
        interestFee: parse6decimal('0.0'),
        oracleFee: parse6decimal('0.0'),
        riskFee: parse6decimal('0.0'),
        positionFee: parse6decimal('0.0'),
        maxPendingGlobal: 5,
        maxPendingLocal: 3,
        settlementFee: parse6decimal('0'),
        makerRewardRate: parse6decimal('0'),
        longRewardRate: parse6decimal('0'),
        shortRewardRate: parse6decimal('0'),
        makerCloseAlways: false,
        takerCloseAlways: false,
        closed: false,
    }

    await market.connect(owner).updateRiskParameter(riskParameter);
    await market.connect(owner).updateParameter(marketParameter);

    setupOracle('100', TIMESTAMP, TIMESTAMP + 100);

    var collateral = parse6decimal('500')
    dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪   collateral.mul(1e12)).returns(true)
    await market.connect(userB).update(userB.address, parse6decimal('50.000'),
↪   0, 0, collateral, false)
    dsu.transferFrom.whenCalledWith(userC.address, market.address,
↪   collateral.mul(1e12)).returns(true)
    await market.connect(userC).update(userC.address, parse6decimal('50.000'),
↪   0, 0, collateral, false)

    var collateral = parse6decimal('10')
    dsu.transferFrom.whenCalledWith(user.address, market.address,
↪   collateral.mul(1e12)).returns(true)
```

```
    await market.connect(user).update(user.address, 0, parse6decimal('10.000'),
↪   0, collateral, false)

    var info = await market.locals(user.address);
    var infoB = await market.locals(userB.address);
    var infoC = await market.locals(userC.address);
    console.log("collateral before liquidation: " + info.collateral + " + " +
↪   infoB.collateral + " + " + infoC.collateral + " = " +
        info.collateral.add(infoB.collateral).add(infoC.collateral));

    setupOracle('100', TIMESTAMP + 100, TIMESTAMP + 200);
    setupOracle('90', TIMESTAMP + 200, TIMESTAMP + 300);
    // liquidate
    const EXPECTED_LIQUIDATION_FEE = parse6decimal('10')
    dsu.transfer.whenCalledWith(liquidator.address,
↪   EXPECTED_LIQUIDATION_FEE.mul(1e12)).returns(true)
    dsu.balanceOf.whenCalledWith(market.address).returns(COLLATERAL.mul(1e12))
    await market.connect(liquidator).update(user.address, 0, 0, 0,
↪   EXPECTED_LIQUIDATION_FEE.mul(-1), true)

    setupOracle('90', TIMESTAMP + 200, TIMESTAMP + 300);
    await market.connect(userB).update(userB.address, 0, 0, 0, 0, false)
    await market.connect(userC).update(userC.address, 0, 0, 0, 0, false)

    var info = await market.locals(user.address);
    var infoB = await market.locals(userB.address);
    var infoC = await market.locals(userC.address);
    console.log("collateral after liquidation: " + info.collateral + " + " +
↪   infoB.collateral + " + " + infoC.collateral + " = " +
        info.collateral.add(infoB.collateral).add(infoC.collateral));
})
```

Console output for the code:

```
collateral before liquidation: 10000000 + 500000000 + 500000000 = 1010000000
collateral after liquidation: -100000080 + 550000000 + 550000000 = 999999920
```

After initial total deposit of $1010, in the end liquidated user will just abandon his account, and remaining user accounts have $550+$550=$1100 but only $1000 funds in the protocol to withdraw.

Code Snippet

When account is liquidated, its collateral becomes negative, but is allowed when protected and the collateral is never reset to 0: https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L497

However, user with negative collateral will simply abandond the account and shortfall will make it impossible to withdraw for the last users in case of bank run.

Tool used

Manual Review

Recommendation

There should be no negative collateral accounts with 0-position and no incentive to cover shortfall. When liquidated, if account is left with negative collateral, the bad debt should be added to the opposite position pnl (long position bad debt should be socialized between short position holders) or maybe to makers pnl only (socialized between makers). The account will have to be left with collateral = 0.

Implementation details for such solution can be tricky due to settlement in the future (pnl is not known at the time of liquidation initiation). Possibly a 2nd step of bad debt liquidation should be added: a keeper will call the user account to socialize bad debt and get some reward for this. Although this is not the best solution, because users who close their positions before the keeper socializes the bad debt, will be able to avoid this social loss. One of the solutions for this will be to introduce delayed withdrawals and delayed socialization (like withdrawals are allowed only after 5 oracle versions and socialization is applied to all positions opened before socialization and still active or closed within 5 last oracle versions), but it will make protocol much more complicated.

Discussion

sherlock-admin

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

141345 commented:

> m

arjun-io

While we like the recommended approach it might be overly cumbersome to implement and still does not fully prevent the shortfall bank run situation. Bad debt/shortfall is necessary in Perennial and should be thoroughly minimized through correct parameter setting.

jacksanford1

From WatchPug:

Acknowledged by Perennial team.

Issue M-5: Market: DoS when stuffed with pending protected positions

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/94

Found by

n33k

Market has no limit on how many protected position updates can be added into pending position update queue. When settling these pending position updates, transations can OOG revert. This fully bricks the protocol and funds will be locked forever.

Vulnerability Detail

In `_invariant`, there is a limit on the number of pending position updates. But for `protected` position updates, `_invariant` returns early and does not trigger this check.

```
function _invariant(
    Context memory context,
    address account,
    Order memory newOrder,
    Fixed6 collateral,
    bool protected
) private view {
    ....

    if (protected) return; // The following invariants do not apply to protected
↪   position updates (liquidations)
    ....
        if (
        context.global.currentId > context.global.latestId +
↪   context.marketParameter.maxPendingGlobal ||
        context.local.currentId > context.local.latestId +
↪   context.marketParameter.maxPendingLocal
    ) revert MarketExceedsPendingIdLimitError();
    ....
}
```

After the `_invariant` check, the postion updates will be added into pending position queues.

```
_invariant(context, account, newOrder, collateral, protected);
```

```
// store
_pendingPosition[context.global.currentId].store(context.currentPosition.global);
_pendingPositions[account][context.local.currentId].store(context.currentPositio
↪  n.local);
```

When the protocol enters next oracle version, the global pending queue `_pendingPosition` will be settled in a loop.

```
function _settle(Context memory context, address account) private {
    ....
    // settle
    while (
        context.global.currentId != context.global.latestId &&
        (nextPosition = _pendingPosition[context.global.latestId +
↪  1].read()).ready(context.latestVersion)
    ) _processPositionGlobal(context, context.global.latestId + 1, nextPosition);
```

The OOG revert happens if there are too many pending position updates.

This revert will happend on every `update` calls because they all need to settle this `_pendingPosition` before update.

```
function update(
    address account,
    UFixed6 newMaker,
    UFixed6 newLong,
    UFixed6 newShort,
    Fixed6 collateral,
    bool protect
) external nonReentrant whenNotPaused {
    Context memory context = _loadContext(account);
    _settle(context, account);
    _update(context, account, newMaker, newLong, newShort, collateral, protect);
    _saveContext(context, account);
}
```

Impact

The protocol will be fully unfunctional and funds will be locked. There will be no recover to this DoS.

A malicious user can tigger this intentionally at very low cost. Alternatively, this can occur during a volatile market period when there are massive liquidations.

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/pack

ages/perennial/contracts/Market.sol#L497

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L505-L508

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L333-L337

Tool used

Manual Review

Recommendation

Either or both,

1. Limit the number of pending protected position updates can be queued in `_invariant`.
2. Limit the number of global pending protected postions can be settled in `_settle`.

Discussion

sherlock-admin

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

141345 commented:

> x

panprog commented:

> medium because it's actually very hard to create many pending positions and at the same time liquidate: new pending position is only created once time advances by the granularity. If previous pending positions are not still settled, this means there were no oracle commits, this means that market price didn't change and accounts can be liquidated the whole time.

syjcnss

Escalate

This should be valid high.

Because the impact is critical and should never happen even under edge cases.

This DoS does not require an attacker. The protocol could enter such state spontaneously if conditions are met.

sherlock-admin2

> Escalate
>
> This should be valid high.
>
> Because the impact is critical and should never happen even under edge cases.
>
> This DoS does not require an attacker. The protocol could enter such state spontaneously if conditions are met.

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.

panprog

Escalate

I agree that this issue is valid. However, it should be medium, not high, because it's actually quite unlikely to enter such state:

1. New pending position (both global and local) is only created when `oracle.current()` timestamp changes. Since it's in granularities, this happens every `granularity` seconds.
2. Whenever an oracle version is commited, all pending positions up to that commit are settled. This means that in order to enter such state - there should be a large number of consecutive requested and uncommited oracle versions.
3. Since such state can only happen when there is no oracle commit for a long time, this means that all this time the same price is used for liquidation, so all accounts potentially liquidatable are liquidatable for this entire time, and for such state to happen, at least 1 account should be liquidated in each different granularity timestamp, which is unlikely - liquidations are more likely to happen all in the first 1-2 oracle versions.

Example of what has to happen to cause this bug: -- no accounts liquidatable t=120: oracle commits new price for `oracle version` = 100 For example, the price change is high and causes mass liquidations t=130: liquidation (new pending position at timestamp 200, 1 pending position) t=140: liquidation (same pending position at timestamp 200, still 1 pending position) t=150: liquidation (same) ... t=210: liquidation (new pending position at timestamp 200, 2 pending positions) t=220: liquidation (same, 2 pending) ... (no oracle commit for timestamp=200) ... t=310: liquidation (new pending position at timestamp 300, 3 pending positions) ... (no oracle commits for neither timestamps 200, nor 300) t=410: liquidation (4 pending) ... (still no oracle commits) t=510: liquidation (5 pending) ... (still no oracle commits) t=610: liquidation (6 pending -> exceeds pending positions limit)

So if pending limit is 5, there should be:

- no oracle commits for at least 4 * granularity seconds
- at least 1 position change request each granularity window
- at least 1 liquidation after 4 * granularity seconds have passed - remember that all 4 * granularity seconds all positions are liquidatable, so they're much more likely to happen earlier rather than after such long period of time.

This will allow to exceed the pending limit. However, the real impact (protocol funds get stuck) will likely occur much later after even more oracle versions and liquidations, because it's unlikely that pending limit is set at the edge of transaction max gas limit.

So given conditions which are quite rare to occur, but possible, this should be medium.

sherlock-admin2

> Escalate
>
> I agree that this issue is valid. However, it should be medium, not high, because it's actually quite unlikely to enter such state:
>
> 1. New pending position (both global and local) is only created when `oracle.current()` timestamp changes. Since it's in granularities, this happens every `granularity` seconds.
> 2. Whenever an oracle version is commited, all pending positions up to that commit are settled. This means that in order to enter such state - there should be a large number of consecutive requested and uncommited oracle versions.
> 3. Since such state can only happen when there is no oracle commit for a long time, this means that all this time the same price is used for liquidation, so all accounts potentially liquidatable are liquidatable for this entire time, and for such state to happen, at least 1 account should be liquidated in each different granularity timestamp, which is unlikely - liquidations are more likely to happen all in the first 1-2 oracle versions.
>
> Example of what has to happen to cause this bug: -- no accounts liquidatable t=120: oracle commits new price for `oracle version` = 100 For example, the price change is high and causes mass liquidations t=130: liquidation (new pending position at timestamp 200, 1 pending position) t=140: liquidation (same pending position at timestamp 200, still 1 pending position) t=150: liquidation (same) ... t=210: liquidation (new pending position at timestamp 200, 2 pending positions) t=220: liquidation (same, 2 pending) ... (no oracle commit for timestamp=200) ... t=310: liquidation (new pending position at timestamp 300, 3 pending

positions) ... (no oracle commits for neither timestamps 200, nor 300) t=410: liquidation (4 pending) ... (still no oracle commits) t=510: liquidation (5 pending) ... (still no oracle commits) t=610: liquidation (6 pending -> exceeds pending positions limit)

So if pending limit is 5, there should be:

- no oracle commits for at least 4 * granularity seconds
- at least 1 position change request each granularity window
- at least 1 liquidation after 4 * granularity seconds have passed - remember that all 4 * granularity seconds all positions are liquidatable, so they're much more likely to happen earlier rather than after such long period of time.

This will allow to exceed the pending limit. However, the real impact (protocol funds get stuck) will likely occur much later after even more oracle versions and liquidations, because it's unlikely that pending limit is set at the edge of transaction max gas limit.

So given conditions which are quite rare to occur, but possible, this should be medium.

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.

syjcnss

I agree with @panprog on his 1,2,3 explainations on why `it's actually quite unlikely to enter such state`.

I think it should be high because there is **NO attacker/special setup**(i.e., `pending limit is set at the edge of transaction max gas limit` in @panprog's comment) **required** for this locking of fund to happen.

panprog

I agree that this state can happen by itself without special setup or attacker, but the likelihood is very low, that's why it's medium. Refer to sherlock rules on how to identify high and medium issues, high issues do not require limiting external conditions, medium issues require certain external conditions or specific states - that's exactly the case for this issue.

https://docs.sherlock.xyz/audits/judging/judging#how-to-identify-a-high-issue

How to identify a high issue:

- **Definite loss of funds without limiting external conditions.**

- Breaks core contract functionality, rendering the protocol/contract useless (should not be easily replaced without loss of funds) and definitely causes significant loss of funds.
- Significant loss of funds/large profit for the attacker at a minimal cost.

How to identify a medium issue:

- **Causes a loss of funds but requires certain external conditions or specific states.**
- Breaks core contract functionality, rendering the contract useless (should not be easily replaced without loss of funds) or leading to unknown potential exploits/loss of funds. Eg: Unable to remove malicious user/collateral from the contract.
- A material loss of funds, no/minimal profit for the attacker at a considerable cost

syjcnss

Don't think `medium issues require certain external conditions or specific states` applies to this one.

The conditions are that untrusted 3rd-party liquidators&keepers behave non-ideally(slow liquidation&price commit) under massive liquidations which should be considered met. And once is all it takes to lock fund.

arjun-io

This is a valid state that can occur so in our view this is a valid issue. However, we agree that there is a very low likelihood that this can occur with correct parameters set for the market

syjcnss

The likelihood is low but the result is untolerable.

The protocol is relying on untrusted 3rd parties to prevent this from happenning. The probability increases under congested network conditions and low liquidator/keeper performance.

Despite the severity of this issue in the contest I suggest to have a fix.

141345

The condition for this to be valid is strict. With long granularity, and large number of pending positions. For a malicious user, there is cost of the position and potential risk. The likelihood for it to happen by user self is even lower. Seems medium severity according to the criteria.

hrishibhat

Result: Medium Unique Agree with the above Escalation and the comment on why this issue should be valid medium. https://github.com/sherlock-audit/2023-07-perennial-judging/issues/94#issuecomment-1697453789

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- syjcnss: accepted
- panprog: accepted

jacksanford1

From WatchPug:

Acknowledged

Issue M-6: It is possible to open and liquidate your own position in 1 transaction to overcome efficiency and liquidity removal limits at almost no cost

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/104

Found by

Emmanuel, panprog

The way the protocol is setup, it is possible to open positions or withdraw collateral up to exactly maintenance limit (some percentage of notional). However, this means that it's possible to be at almost liquidation level intentionally and moreover, the current oracle setup allows to open and immediately liquidate your own position in 1 transaction, effectively bypassing efficiency and liquidity removal limits, paying only the keeper (and possible position open/close) fees, causing all kinds of malicious activity which can harm the protocol.

Vulnerability Detail

The user can liquidate his own position with 100% guarantee in 1 transaction by following these steps:

1. It can be done on existing position or on a new position
2. Record Pyth oracle prices with signatures until you encounter a price which is higher (or lower, depending on your position direction) than latest oracle version price by any amount.
3. In 1 transaction do the following: 3.1. Make the position you want to liquidate at exactly the edge of liquidation: withdraw maximum allowed amount or open a new position with minimum allowed collateral 3.2. Commit non-requested oracle version with the price recorded earlier (this price makes the position liquidatable) 3.3. Liquidate your position (it will be allowed, because the position generates a minimum loss due to price change and becomes liquidatable)

Since all liquidation fee is given to user himself, liquidation of own position is almost free for the user (only the keeper and position open/close fee is paid if any).

Impact

There are different malicious actions scenarios possible which can abuse this issue and overcome efficiency and liquidity removal limitations (as they're ignored when liquidating positions), such as:

- Open large maker and long or short position, then liquidate maker to cause mismatch between long/short and maker (socialize positions). This will cause some chaos in the market, disbalance between long and short profit/loss and users will probably start leaving such chaotic market, so while this attack is not totally free, it's cheap enough to drive users away from competition.
- Open large maker, wait for long and/or short positions from normal users to accumulate, then liquidate most of the large maker position, which will drive taker interest very high and remaining small maker position will be able to accumulate big profit with a small risk.
- Just open long/short position from different accounts and wait for the large price update and frontrun it by withdrawing max collateral from the position which will be in a loss, and immediately liquidate it in the same transaction: with large price update one position will be liquidated with bad debt while the other position will be in a large profit, total profit from both positions will be positive and basically risk-free, meaning it's at the expense of the other users. While this strategy is possible to do on its own, liquidation in the same transaction allows it to be more profitable and catch more opportunities, meaning more damage to the other protocol users.

The same core reason can also cause unsuspecting user to be unexpectedly liquidated in the following scenario:

1. User opens position (10 ETH long at $1000, with $10000 collateral). User is choosing very safe leverage = 1. Market maintenance is set to 20% (max leverage = 5)
2. Some time later the price is still $1000 and user decides to close most of his position and withdraw collateral, so he reduces his position to 2 ETH long and withdraws $8000 collateral, leaving his position with $2000 collateral. It appears that the user is at the safe leverage = 1 again.
3. Right in the same block the liquidator commits non-requested oracle with a price $999.999 and immediately liquidates the user.

The user is unsuspectedly liquidated even though he thought that he was at leverage = 1. But since collateral is withdrawn immediately, but position changes only later, user actually brought his position to max leverage and got liquidated. While this might be argued to be the expected behavior, it might still be hard to understand and unintuitive for many users, so it's better to prevent such situation from happening and the fix is the same as the one to fix self-liquidations.

Proof of concept

The scenario of liquidating unsuspecting user is demonstrated in the test, add this to test/unit/market/Market.test.ts:

```
it('panprog liquidate unsuspecting user / self in 1 transaction', async () => {

    function setupOracle(price: string, timestamp : number, nextTimestamp :
↪   number) {
        const oracleVersion = {
        price: parse6decimal(price),
        timestamp: timestamp,
        valid: true,
        }
        oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
        oracle.status.returns([oracleVersion, nextTimestamp])
        oracle.request.returns()
    }

    var riskParameter = {
        maintenance: parse6decimal('0.2'),
        takerFee: parse6decimal('0.00'),
        takerSkewFee: 0,
        takerImpactFee: 0,
        makerFee: parse6decimal('0.00'),
        makerImpactFee: 0,
        makerLimit: parse6decimal('1000'),
        efficiencyLimit: parse6decimal('0.2'),
        liquidationFee: parse6decimal('0.50'),
        minLiquidationFee: parse6decimal('10'),
        maxLiquidationFee: parse6decimal('1000'),
        utilizationCurve: {
        minRate: parse6decimal('0.0'),
        maxRate: parse6decimal('1.00'),
        targetRate: parse6decimal('0.10'),
        targetUtilization: parse6decimal('0.50'),
        },
        pController: {
        k: parse6decimal('40000'),
        max: parse6decimal('1.20'),
        },
        minMaintenance: parse6decimal('10'),
        virtualTaker: parse6decimal('0'),
        staleAfter: 14400,
        makerReceiveOnly: false,
    }
    var marketParameter = {
        fundingFee: parse6decimal('0.0'),
        interestFee: parse6decimal('0.0'),
        oracleFee: parse6decimal('0.0'),
        riskFee: parse6decimal('0.0'),
```

```
        positionFee: parse6decimal('0.0'),
        maxPendingGlobal: 5,
        maxPendingLocal: 3,
        settlementFee: parse6decimal('0'),
        makerRewardRate: parse6decimal('0'),
        longRewardRate: parse6decimal('0'),
        shortRewardRate: parse6decimal('0'),
        makerCloseAlways: false,
        takerCloseAlways: false,
        closed: false,
    }

    await market.connect(owner).updateRiskParameter(riskParameter);
    await market.connect(owner).updateParameter(marketParameter);

    setupOracle('100', TIMESTAMP, TIMESTAMP + 100);

    var collateral = parse6decimal('1000')
    dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪   collateral.mul(1e12)).returns(true)
    await market.connect(userB).update(userB.address, parse6decimal('10.000'),
↪   0, 0, collateral, false)

    var collateral = parse6decimal('100')
    dsu.transferFrom.whenCalledWith(user.address, market.address,
↪   collateral.mul(1e12)).returns(true)
    await market.connect(user).update(user.address, 0, parse6decimal('1.000'),
↪   0, collateral, false)

    // settle
    setupOracle('100', TIMESTAMP + 100, TIMESTAMP + 200);
    await market.connect(userB).update(userB.address, parse6decimal('10.000'),
↪   0, 0, 0, false)
    await market.connect(user).update(user.address, 0, parse6decimal('1.000'),
↪   0, 0, false)

    // withdraw
    var collateral = parse6decimal('800')
    dsu.transfer.whenCalledWith(userB.address,
↪   collateral.mul(1e12)).returns(true)
    await market.connect(userB).update(userB.address, parse6decimal('2.000'), 0,
↪   0, collateral.mul(-1), false)

    // liquidate unsuspecting user
    setupOracle('100.01', TIMESTAMP + 150, TIMESTAMP + 200);
    const EXPECTED_LIQUIDATION_FEE = parse6decimal('100.01')
    dsu.transfer.whenCalledWith(liquidator.address,
↪   EXPECTED_LIQUIDATION_FEE.mul(1e12)).returns(true)
```


```
    dsu.balanceOf.whenCalledWith(market.address).returns(COLLATERAL.mul(1e12))
    await market.connect(liquidator).update(userB.address, 0, 0, 0,
↪   EXPECTED_LIQUIDATION_FEE.mul(-1), true)

    setupOracle('100.01', TIMESTAMP + 200, TIMESTAMP + 300);
    await market.connect(userB).update(userB.address, 0, 0, 0, 0, false)

    var info = await market.locals(userB.address);
    var pos = await market.positions(userB.address);
    console.log("Liquidated maker: collateral = " + info.collateral + " maker =
↪   " + pos.maker);

})
```

Console output for the code:

```
Liquidated maker: collateral = 99980000 maker = 0
```

Self liquidation is the same, just the liquidator does this in 1 transaction and is owned by userB.

Code Snippet

Account solvency is calculated as meeting the minimum collateral of maintenance (percentage of notional): https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/types/Position.sol#L305

It is possible to bring user to exactly the edge of liquidation, when minimum loss makes him liquidatable.

Tool used

Manual Review

Recommendation

Industry standard is to have initial margin (margin required to open position or withdraw collateral) and maintenance margin (margin required to keep the position solvent). Initial margin > maintenance margin and serves exactly for the reason to prevent users from being close to liquidation, intentional or not. I suggest to implement initial margin as a measure to prevent such self liquidation or unsuspected user liquidations. This will improve user experience (remove a lot of surprise liquidations) and will also improve security by disallowing intentional liquidations and cheaply overcoming the protocol limits such as efficiency limit: intentional liquidations are never good for the protocol as they're most often

malicious, so having the ability to liquidate yourself in 1 transaction should definetely be prohibited.

Discussion

sherlock-admin

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

141345 commented:

> m

arjun-io

The self liquidations do seem possible here, we'll look further into the downstream impacts to figure out any fixes we want to implement.

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/92

MLON33

From WatchPug,

Fixed.

Issue M-7: update() wrong privilege control

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/121

Found by

bin2chen `oracle.update()` wrong privilege control lead to `OracleFactory.update()` unable to add `oracleProvider`

Vulnerability Detail

in `OracleFactory.update()` will call `oracle.update()`

```
contract OracleFactory is IOracleFactory, Factory {
...
    function update(bytes32 id, IOracleProviderFactory factory) external
↪   onlyOwner {
        if (!factories[factory]) revert OracleFactoryNotRegisteredError();
        if (oracles[id] == IOracleProvider(address(0))) revert
↪   OracleFactoryNotCreatedError();

        IOracleProvider oracleProvider = factory.oracles(id);
        if (oracleProvider == IOracleProvider(address(0))) revert
↪   OracleFactoryInvalidIdError();

        IOracle oracle = IOracle(address(oracles[id]));
@>      oracle.update(oracleProvider);
    }
```

But `oracle.update()` permission is needed for `OracleFactory.owner()` and not `OracleFactory` itself.

```
@>  function update(IOracleProvider newProvider) external onlyOwner {
        _updateCurrent(newProvider);
        _updateLatest(newProvider.latest());
    }

    modifier onlyOwner {
@>      if (msg.sender != factory().owner()) revert
↪   InstanceNotOwnerError(msg.sender);
        _;
    }
```

This results in `OracleFactory` not being able to do `update()`. Suggest changing the limit of `oracle.update()` to `factory()`.

Impact

`OracleFactory.update()` unable to add `IOracleProvider`

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/OracleFactory.sol#L81

Tool used

Manual Review

Recommendation

```
contract Oracle is IOracle, Instance {
...

-   function update(IOracleProvider newProvider) external onlyOwner {
+   function update(IOracleProvider newProvider) external {
+       require(msg.sender == factory(),"invalid sender");
        _updateCurrent(newProvider);
        _updateLatest(newProvider.latest());
    }
```

Discussion

sherlock-admin

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

141345 commented:

> m

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/81

MLON33

From WatchPug,

Fixed.

Issue M-8: `_accumulateFunding()` maker will get the wrong amount of funding fee.

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/139

Found by

WATCHPUG

Vulnerability Detail

The formula that calculates the amount of funding in `Version#_accumulateFunding()` on the maker side is incorrect. This leads to an incorrect distribution of funding between the minor and the maker's side.

```
// Redirect net portion of minor's side to maker
if (fromPosition.long.gt(fromPosition.short)) {
    fundingValues.fundingMaker =
↪   fundingValues.fundingShort.mul(Fixed6Lib.from(fromPosition.skew().abs()));
    fundingValues.fundingShort =
↪   fundingValues.fundingShort.sub(fundingValues.fundingMaker);
}
if (fromPosition.short.gt(fromPosition.long)) {
    fundingValues.fundingMaker =
↪   fundingValues.fundingLong.mul(Fixed6Lib.from(fromPosition.skew().abs()));
    fundingValues.fundingLong =
↪   fundingValues.fundingLong.sub(fundingValues.fundingMaker);
}
```

PoC

Given:

- long/major: 1000
- short/minor: 1
- maker: 1

Then:

1. skew(): 999/1000
2. fundingMaker: 0.999 of the funding
3. fundingShort: 0.001 of the funding

While the maker only matches for `1` of the major part and contributes to half of the total short side, it takes the entire funding.

Impact

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/types/Version.sol#L207-L215

Tool used

Manual Review

Recommendation

The correct formula to calculate the amount of funding belonging to the maker side should be:

```
fundingMakerRatio = min(maker, major - minor) / min(major, minor + maker)
fundingMaker = fundingMakerRatio * fundingMinor
```

Discussion

sherlock-admin

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

141345 commented:

> m

panprog commented:

> medium because incorrect result only starts appearing if abs(long-short) > maker and the larger the difference, the more incorrect the split of funding is. But this situation is exceptional case, most of the time abs(long-short) < maker due to efficiency and liquidity limits

arjun-io

We'd like to re-open this as it does appear to be a valid issue. Medium severity seems correct here

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/64

MLON33

From WatchPug,

Fixed alongside with #180.

Issue M-9: `OracleVersion latestVersion` of `Oracle.status()` may go backwards when updating to a new oracle provider and result in wrong settlement in `_processPositionLocal()`.

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/145

Found by

WATCHPUG

Vulnerability Detail

This is because when `Oracle.update(newProvider)` is called, there is no requirement that `newProvider.latest().timestamp > oldProvider.latest().timestamp`.

During the `processLocal`, encountering a non-existing version will result in using 0 as the `makerValue`, `longValue`, and `shortValue` to settle PNL, causing the user's collateral to be deducted incorrectly.

This is because L350 is skipped (as the global has been settled to a newer timestamp), and L356 enters the if branch.

PoC

Given:

- At 13:40, The `latest().timestamp` of oracleProvider1 is 13:30

When:

- At 13:40, market.update(account1, ...)
  - Store `_versions[13:00]` in L337
  - Store `_versions[13:30]` in L353
- At 13:41, `oracle.update(oracleProvider2)` (note: The current `latest().timestamp` of oracleProvider2 is 13:20)
- `market.update(account2) -> _settle()`, L350 is skipped; L356 `13:20 > 13:00`, enters `_processPositionLocal()`:
  - L436, `nextPosition.timestamp == 13:20`, `version` is empty;
  - L440, `context.local.accumulate` with empty `version` will result in wrong PNL.

Impact

Code Snippet

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial-oracle/contracts/Oracle.sol#L106-L117

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L327-L364

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L390-L423

https://github.com/sherlock-audit/2023-07-perennial/blob/main/perennial-v2/packages/perennial/contracts/Market.sol#L430-L457

Tool used

Manual Review

Recommendation

Consider requireing `newProvider.latest().timestamp > oldProvider.latest().timestamp` in `Oracle.update(newProvider)`.

Discussion

sherlock-admin

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

141345 commented:

> d

panprog commented:

> invalid because this is owner error who is trusted, also there is no real impact (update will simply revert if the time goes backward)

panprog

This should be valid medium, even though not escalated. POC:

```
it('latest.timestamp moving back in time', async () => {

  function setupOracle(price: string, timestamp : number, nextTimestamp :
↪  number) {
    const oracleVersion = {
      price: parse6decimal(price),
      timestamp: timestamp,
```

```
      valid: true,
    }
    oracle.at.whenCalledWith(oracleVersion.timestamp).returns(oracleVersion)
    oracle.status.returns([oracleVersion, nextTimestamp])
    oracle.request.returns()
  }

  var marketParameter = {
    fundingFee: parse6decimal('0.1'),
    interestFee: parse6decimal('0.1'),
    oracleFee: parse6decimal('0.1'),
    riskFee: parse6decimal('0.1'),
    positionFee: parse6decimal('0.1'),
    maxPendingGlobal: 5,
    maxPendingLocal: 3,
    settlementFee: parse6decimal('0'),
    makerRewardRate: parse6decimal('0.0'),
    longRewardRate: parse6decimal('0.0'),
    shortRewardRate: parse6decimal('0.0'),
    makerCloseAlways: false,
    takerCloseAlways: false,
    closed: false,
  }

  await market.connect(owner).updateParameter(marketParameter);

  setupOracle('100', TIMESTAMP, TIMESTAMP + 100);

  var collateral = parse6decimal('1000')
  dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪  collateral.mul(1e12)).returns(true)
  await market.connect(userB).update(userB.address, parse6decimal('10.000'), 0,
↪  0, collateral, false)

  var collateral = parse6decimal('120')
  dsu.transferFrom.whenCalledWith(user.address, market.address,
↪  collateral.mul(1e12)).returns(true)
  await market.connect(user).update(user.address, 0, parse6decimal('1.000'), 0,
↪  collateral, false)

  // open position
  setupOracle('100', TIMESTAMP + 100, TIMESTAMP + 200);
  await market.connect(user).update(user.address, 0, parse6decimal('1.000'), 0,
↪  0, false)

  var info = await market.locals(user.address);
  var pos = await market.positions(user.address);
```

```
  console.log("after open (price=100): user collateral = " + info.collateral + "
↪  long = " + pos.long);

  // accumulate some pnl
  setupOracle('90', TIMESTAMP + 200, TIMESTAMP + 300);
  await market.connect(user).update(user.address, 0, parse6decimal('1.000'), 0,
↪  0, false)

  var info = await market.locals(user.address);
  var pos = await market.positions(user.address);
  var ver = await market.versions(TIMESTAMP + 200);
  console.log("after settle pnl (price=90): user collateral = " +
↪  info.collateral + " long = " + pos.long + " ver_longValue: " + ver.longValue
↪  + " ver_makerValue: " + ver.makerValue);

  // add collateral only
  setupOracle('90', TIMESTAMP + 300, TIMESTAMP + 400);
  dsu.transferFrom.whenCalledWith(userB.address, market.address,
↪  collateral.mul(1e12)).returns(true)
  await market.connect(userB).update(userB.address, parse6decimal('10.000'), 0,
↪  0, collateral, false)

  // oracle.latest moves back in time
  setupOracle('89', TIMESTAMP + 290, TIMESTAMP + 400);
  await market.connect(user).update(user.address, 0, parse6decimal('1.000'), 0,
↪  0, false)

  var info = await market.locals(user.address);
  var pos = await market.positions(user.address);
  console.log("after move back in time (price=89): user collateral = " +
↪  info.collateral + " long = " + pos.long);

  setupOracle('89', TIMESTAMP + 400, TIMESTAMP + 500);
  await market.connect(user).update(user.address, 0, parse6decimal('1.000'), 0,
↪  0, false)
  setupOracle('89', TIMESTAMP + 500, TIMESTAMP + 600);
  await market.connect(user).update(user.address, 0, parse6decimal('1.000'), 0,
↪  0, false)

  var info = await market.locals(user.address);
  var pos = await market.positions(user.address);
  console.log("User settled (price=89): collateral = " + info.collateral + "
↪  long = " + pos.long);

})
```

Console output:

```
after open (price=100): user collateral = 120000000 long = 1000000
after settle pnl (price=90): user collateral = 109999994 long = 1000000
↪  ver_longValue: -10000006 ver_makerValue: 1000000
after move back in time (price=89): user collateral = 120000000 long = 1000000
User settled (price=89): collateral = 108999975 long = 1000000
```

panprog

Medium, not high, because:

1. Can only happen during oracle provider switch (which is a rare admin event by itself)
2. Very specific condition must be met: previous provider must be commited unrequested past the last requested AND new provider must be commited unrequested with newProvider.latest.timestamp < previousProvider.latest.timestamp AND there should be a user who has pending position at a timestamp > previousProvider.latest.timestamp (no position change to not create new request).
3. Only possible if accumulated reward is 0, meaning the market must have all reward rates = 0 for the entire market lifetime (otherwise it will revert when trying to accumulate reward, which is unsigned).
4. The effect is only temporary because the "0" version is invalid and is discarded once a valid version appears (any commit is done for new provider)

hrishibhat

Considering this issue a Unique Medium based on the above comments

kbrizzle

Looks like we missed this one since it was reopened last minute. After an initial review we think this is a false-positive, but please check our reasoning and let us know if you still see an issue.

The actual potential bug here is an incorrect implementation of the `IOracleProvider` interface in `Oracle`. The extra information about what happens inside the `Market` if latest can be out of order is not relevant, as many things can go wrong if that invariant is not upheld in the implementation.

So what we'd be looking for here for a valid bug is a specific case where an `Oracle` updating from one sub-oracle to another sub-oracle would cause the `latest()` to return a value lower than it previously did.

I ran through the case that I think you've outlined here (though I may have translate it incorrectly) and I don't see a way for it to bypass the `_latestStale` check, which is specifically designed to handle this case.

Case

```
oracle 0 -> current: 330, latest: 330
oracle 1 -> current: 330, latest: 320
```

Assuming `Oracle.update()` was called at `330`, once in the state above this check will pass, but this check will still fail. This keeps the `Oracle` pointed at `oracle 0` until the latest clears as expected.

Let us know if there's an example here you can show us that does in fact bypass the `_latestStale` function.

MLON33

From WatchPug,

No PR attached.

panprog

> Let us know if there's an example here you can show us that does in fact bypass the `_latestStale` function.

`oracles[global.latest].timestamp` - is the last time oracle.request() was called for a provider. Since provider can commit unrequested, `provider1.latest.timestamp` can be greater than `oracles[global.latest].timestamp`. So the following values can happen:

- `oracles[global.latest].timestamp` = 320
- `provider1.latest.timestamp` = 330 (commited unrequested)
- `provider2.latest.timestamp` = 325 (also commited unrequested)

To cause these values, the following actions might happen: t=320: `Oracle.request()` called [`oracles[global.latest].timestamp` = 320] t=340: `provider1.commitRequested()` called [`provider1.latest.timestamp` = 320] t=350: `provider1.commit(330)` called [`provider1.latest.timestamp` = 330] t=350: `provider2.commit(325)` called [`provider2.latest.timestamp` = 325] t=350: `Oracle.update(provider2)` called

In such situations both checks in `_latestStale()` will pass:

1. 320 > 330: condition not satisfied
2. 320 >= 325: condition not satisfied So, the provider2 will be set immediately, and so `Oracle.lastest()` will return `provider2.latest()` which is earlier than `provider1.latest()`

kbrizzle

Thanks @panprog, this is exactly what we needed will get back on a fix.

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/99

Issue M-10: Drained oracle fees from market by depositing and withdrawing immediately without triggering settlement fees

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/153

Found by

0x73696d616f The oracle fee can be drained, as requests can be made without paying fees by depositing and withdrawing immediately, leading to theft of yield to the `keeper` and potentially a DoS in the system. The DoS happens because the oracle version must be increased to trigger the settlements (global and local), such that a `keeper` amount must be available prior to the new version oracle request. This `keeper` amount would not be available as attackers would have drained the fees before any settlement occurs.

Vulnerability Detail

`Market` advances in the id of the Global and Local states by fetching `latestVersion` and `currentTimestamp` from the oracle, increasing it if there is an update.

When a new position is updated by calling `update()`, if the order is not empty (when there is a modification in the `maker` , `long` or `short` amounts), it requests a new version from the oracle. This means that users can trigger a request with the smallest position possible (1), not paying any fees.

Fetching a price in the oracle is expensive, thus `Perennial` attributes an `oracleFee` to the oracle, which is then fed to the `keeper` for commiting prices in the oracle. Notice that anyone can be the `keeper`, the only requirement is to submit a valid price to the oracle.

In the oracle, the incentive is only paid if there was a previous request, most likely from `Market` (can be any authorized entity). As the `oracleFee` is only increased on settlements, an oracle request can be triggered at any block (only 1 request per block is allowed) by depositing and withdrawing in the same block, without providing any settlement fee.

Thus, this mechanism can be exploited to give maximum profit to the `keeper`, which would force the protocol to manually add fees to the oracle or be DoSed (could be both).

Impact

Theft of yield and protocol funds by abusing the `keeper` role and DoS of the `Market`.

Code Snippet

Added the following test to `Market.test.ts`, proving that the request can be triggered without paying any fees. The attacker would then proceed to commit a price to the oracle and get the fees (possible at every block), until there are no more fees in the `Market`.

```
it.only('POC opens and closes the position to trigger an oracle request without
↪  paying any fee', async () => {
  const dustCollateral = parse6decimal("100");
  const dustPosition = parse6decimal ("0.000001");
  dsu.transferFrom.whenCalledWith(user.address, market.address,
↪  dustCollateral.mul(1e12)).returns(true)

  await expect(market.connect(user).update(user.address, dustPosition, 0, 0,
↪  dustCollateral, false))
    .to.emit(market, 'Updated')
    .withArgs(user.address, ORACLE_VERSION_2.timestamp, dustPosition, 0, 0,
↪  dustCollateral, false)

  expectLocalEq(await market.locals(user.address), {
    currentId: 1,
    latestId: 0,
    collateral: dustCollateral,
    reward: 0,
    protection: 0,
  })
  expectPositionEq(await market.positions(user.address), {
    ...DEFAULT_POSITION,
    timestamp: ORACLE_VERSION_1.timestamp,
  })
  expectPositionEq(await market.pendingPositions(user.address, 1), {
    ...DEFAULT_POSITION,
    timestamp: ORACLE_VERSION_2.timestamp,
    maker: dustPosition,
    delta: dustCollateral,
  })
  expectGlobalEq(await market.global(), {
    currentId: 1,
    latestId: 0,
    protocolFee: 0,
    oracleFee: 0,
    riskFee: 0,
    donation: 0,
  })
  expectPositionEq(await market.position(), {
    ...DEFAULT_POSITION,
    timestamp: ORACLE_VERSION_1.timestamp,
```

```
  })
  expectPositionEq(await market.pendingPosition(1), {
    ...DEFAULT_POSITION,
    timestamp: ORACLE_VERSION_2.timestamp,
    maker: dustPosition,
  })
  expectVersionEq(await market.versions(ORACLE_VERSION_1.timestamp), {
    makerValue: { _value: 0 },
    longValue: { _value: 0 },
    shortValue: { _value: 0 },
    makerReward: { _value: 0 },
    longReward: { _value: 0 },
    shortReward: { _value: 0 },
  })

  dsu.transfer.whenCalledWith(user.address,
↪  dustCollateral.mul(1e12)).returns(true)
  await expect(market.connect(user).update(user.address, 0, 0, 0,
↪  dustCollateral.mul(-1), false))
    .to.emit(market, 'Updated')
    .withArgs(user.address, ORACLE_VERSION_2.timestamp, 0, 0, 0,
↪  dustCollateral.mul(-1), false)

    expect(oracle.request).to.have.been.calledWith(user.address)
})
```

Tool used

Vscode, Hardhat, Manual Review

Recommendation

Whitelist the keeper role to prevent malicious users from figuring out ways to profit from the incentive mechanism. Additionally, the whitelisted keppers could skip oracle requests if they don't contribute to settlements (when there are no orders to settle), to ensure that funds are always available.

Discussion

sherlock-admin

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

141345 commented:

> x

panprog commented:

invalid because deposit will add pending keeper fees and minCollateral will ensure that at least minCollateral can not be withdrawn before position is closed

0x73696d616f

Escalate As long as there are no new positions to settle and the oracle version/timestamp is not updated, it's possible to keep opening and closing positions at each block, triggering consecutive requests, without paying any fees to the protocol.

Then, after a few requests are triggered or someone else either opens a position or commits a new price, the attacker can commit all the previous requests. The only requirement is that the publish time of the requests are within the `MIN_VALID_TIME_AFTER_VERSION` and `MAX_VALID_TIME_AFTER_VERSION` window.

This would pay the attacker the incentives and the profit depends on the specific math (considering the gas fees). Additionally, over a long enough period, it could drain the protocol of keeper fees, which would lead to DoS due to lack of fees to commit new prices.

I tweaked the POC a bit to show how several requests can be made consecutively without paying any fees to the protocol. The positions are opened and closed in the `loop` (incrementing the block.timestamp at each iteration), triggering several requests. The next step would be to commit these requests in the oracle, but that should not require a POC.

```
it.only('POC opens and closes the position to trigger an oracle request without
↪  paying any fee', async () => {
  const dustCollateral = parse6decimal("100");
  const dustPosition = parse6decimal ("0.000001");

  for (let i = 0; i < 5; i++) {
    dsu.transferFrom.whenCalledWith(user.address, market.address,
↪  dustCollateral.mul(1e12)).returns(true)

    await expect(market.connect(user).update(user.address, dustPosition, 0, 0,
↪  dustCollateral, false))
      .to.emit(market, 'Updated')
      .withArgs(user.address, ORACLE_VERSION_2.timestamp, dustPosition, 0, 0,
↪  dustCollateral, false)

    expectLocalEq(await market.locals(user.address), {
      currentId: 1,
      latestId: 0,
      collateral: dustCollateral,
      reward: 0,
      protection: 0,
    })
```

```
    expectPositionEq(await market.positions(user.address), {
      ...DEFAULT_POSITION,
      timestamp: ORACLE_VERSION_1.timestamp,
    })
    expectPositionEq(await market.pendingPositions(user.address, 1), {
      ...DEFAULT_POSITION,
      timestamp: ORACLE_VERSION_2.timestamp,
      maker: dustPosition,
      delta: dustCollateral,
    })
    expectGlobalEq(await market.global(), {
      currentId: 1,
      latestId: 0,
      protocolFee: 0,
      oracleFee: 0,
      riskFee: 0,
      donation: 0,
    })
    expectPositionEq(await market.position(), {
      ...DEFAULT_POSITION,
      timestamp: ORACLE_VERSION_1.timestamp,
    })
    expectPositionEq(await market.pendingPosition(1), {
      ...DEFAULT_POSITION,
      timestamp: ORACLE_VERSION_2.timestamp,
      maker: dustPosition,
    })
    expectVersionEq(await market.versions(ORACLE_VERSION_1.timestamp), {
      makerValue: { _value: 0 },
      longValue: { _value: 0 },
      shortValue: { _value: 0 },
      makerReward: { _value: 0 },
      longReward: { _value: 0 },
      shortReward: { _value: 0 },
    })

    dsu.transfer.whenCalledWith(user.address,
↪   dustCollateral.mul(1e12)).returns(true)
    await expect(market.connect(user).update(user.address, 0, 0, 0,
↪   dustCollateral.mul(-1), false))
      .to.emit(market, 'Updated')
      .withArgs(user.address, ORACLE_VERSION_2.timestamp, 0, 0, 0,
↪   dustCollateral.mul(-1), false)

      expect(oracle.request).to.have.been.calledWith(user.address)

    await time.increase(1)
  }
```

```
})
```

sherlock-admin2

> Escalate As long as there are no new positions to settle and the oracle version/timestamp is not updated, it's possible to keep opening and closing positions at each block, triggering consecutive requests, without paying any fees to the protocol.
>
> Then, after a few requests are triggered or someone else either opens a position or commits a new price, the attacker can commit all the previous requests. The only requirement is that the publish time of the requests are within the `MIN_VALID_TIME_AFTER_VERSION` and `MAX_VALID_TIME_AFTER_VERSION` window.
>
> This would pay the attacker the incentives and the profit depends on the specific math (considering the gas fees). Additionally, over a long enough period, it could drain the protocol of keeper fees, which would lead to DoS due to lack of fees to commit new prices.
>
> I tweaked the POC a bit to show how several requests can be made consecutively without paying any fees to the protocol. The positions are opened and closed in the `loop` (incrementing the block.timestamp at each iteration), triggering several requests. The next step would be to commit these requests in the oracle, but that should not require a POC.

```
it.only('POC opens and closes the position to trigger an oracle request
↪  without paying any fee', async () => {
  const dustCollateral = parse6decimal("100");
  const dustPosition = parse6decimal ("0.000001");

  for (let i = 0; i < 5; i++) {
    dsu.transferFrom.whenCalledWith(user.address, market.address,
↪  dustCollateral.mul(1e12)).returns(true)

    await expect(market.connect(user).update(user.address, dustPosition, 0,
↪  0, dustCollateral, false))
      .to.emit(market, 'Updated')
      .withArgs(user.address, ORACLE_VERSION_2.timestamp, dustPosition, 0,
↪  0, dustCollateral, false)

    expectLocalEq(await market.locals(user.address), {
      currentId: 1,
      latestId: 0,
      collateral: dustCollateral,
      reward: 0,
      protection: 0,
    })
```

```
    expectPositionEq(await market.positions(user.address), {
      ...DEFAULT_POSITION,
      timestamp: ORACLE_VERSION_1.timestamp,
    })
    expectPositionEq(await market.pendingPositions(user.address, 1), {
      ...DEFAULT_POSITION,
      timestamp: ORACLE_VERSION_2.timestamp,
      maker: dustPosition,
      delta: dustCollateral,
    })
    expectGlobalEq(await market.global(), {
      currentId: 1,
      latestId: 0,
      protocolFee: 0,
      oracleFee: 0,
      riskFee: 0,
      donation: 0,
    })
    expectPositionEq(await market.position(), {
      ...DEFAULT_POSITION,
      timestamp: ORACLE_VERSION_1.timestamp,
    })
    expectPositionEq(await market.pendingPosition(1), {
      ...DEFAULT_POSITION,
      timestamp: ORACLE_VERSION_2.timestamp,
      maker: dustPosition,
    })
    expectVersionEq(await market.versions(ORACLE_VERSION_1.timestamp), {
      makerValue: { _value: 0 },
      longValue: { _value: 0 },
      shortValue: { _value: 0 },
      makerReward: { _value: 0 },
      longReward: { _value: 0 },
      shortReward: { _value: 0 },
    })

    dsu.transfer.whenCalledWith(user.address,
↪   dustCollateral.mul(1e12)).returns(true)
    await expect(market.connect(user).update(user.address, 0, 0, 0,
↪   dustCollateral.mul(-1), false))
      .to.emit(market, 'Updated')
      .withArgs(user.address, ORACLE_VERSION_2.timestamp, 0, 0, 0,
↪   dustCollateral.mul(-1), false)

      expect(oracle.request).to.have.been.calledWith(user.address)

    await time.increase(1)
  }
```


```
})
```

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

arjun-io

This is a great find - while the keeper and position fees are correctly accounted for in most cases, this single version open and close *does not* correctly debit these fees and require the collateral balance to be higher than the fee amount. Thank you for the thorough test case as well. We will fix this

nevillehuang

Escalate

Confirmed by sponsor above, fees are not correctly debited for single version open and close.

sherlock-admin2

> Escalate
>
> Confirmed by sponsor above, fees are not correctly debited for single version open and close.

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.

141345

The severity of medium seems more appropriate.

Because according to sherlock's HM criteria:

> Medium: viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future.
>
> High: This vulnerability would result in a material loss of funds, and the cost of the attack is low.

Here:

- The loss is on oracle fee.

- Each time the loss is capped by one time keeper fee, not significant compared to the trading volume.
- And there is cost to fetch valid price data for the attacker.

In summary, the loss is limited with cost, medium might be suffice.

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/88

hrishibhat

Result: Medium Unique Considering this issue a valid medium based on the escalation and Sponsor's comment

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- nevillehuang: accepted

MLON33

From WatchPug,

Fixed.

Issue M-11: [Perennial Self Report] Fix non-requested commits after oracle grace period

Source: https://github.com/sherlock-audit/2023-07-perennial-judging/issues/177

Found by

Protocol Team Medium

When a requested version was unavailable, non-requested versions would be blocked from being to be committed until a new requested version was committed. This could prevent liquidations from occurring.

https://github.com/equilibria-xyz/perennial-v2/pull/58

Discussion

hrishibhat

This issue is not included in the contest pool rewards

arjun-io

Fixed: https://github.com/equilibria-xyz/perennial-v2/pull/58

MLON33

From WatchPug,

https://github.com/equilibria-xyz/perennial-v2/blob/f216b2fd0ec45ea22b443a00fdfe2257f803ce7c/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L89-L102

https://github.com/equilibria-xyz/perennial-v2/blob/f216b2fd0ec45ea22b443a00fdfe2257f803ce7c/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L125-L203

`latest()` may unexpectedly go back in time.

Given:

versionList[18]: 18:59:58 versionList[19]: 19:00:00 nextVersionIndexToCommit: 11
When:

- `commit({versionIndex: 19, oracleVersion: 18:59:59, updateData: (pyth data of 19:00:03)})`
  - At L180, `oracleVersion == versionList[versionIndex]` i.e., `18:59:59 == 19:00:00` is false, so it won't enter `commitRequested()`

    - `commit()` does not have a check like `commitRequested()` L153-L157, so it can skip `versionIndex: 18`, even if `updateData` is a valid price for `versionIndex: 18`
- At this point, latest() returns OracleVersion(timestamp: 18:59:59, ...)
- `commitRequested({versionIndex: 18, updateData: (pyth data of 19:00:04)})`
    - `commitRequested()` does not have a check like `commit()` L199's newlatestVersion > oldlatestVersion (i.e., versionToCommit > _latestVersion)
- At this point, `latest()` returns OracleVersion(timestamp: 18:59:58, ...)

Recommendation

In addition to the invariant of `newPrice.publishTime > lastCommittedPublishTime`, make sure `newLatestVersion > oldLatestVersion`.

arjun-io

From WatchPug,

https://github.com/equilibria-xyz/perennial-v2/blob/f216b2fd0ec45ea22b443a00fdfe2257f803ce7c/packages/perennial-oracle/contracts/pyth/PythOracle.sol#L89-L102

`latest()` may unexpectedly go back in time.

Given:

versionList[18]: 18:59:58 versionList[19]: 19:00:00
nextVersionIndexToCommit: 11 When:

- `commit({versionIndex: 19, oracleVersion: 18:59:59, updateData: (pyth data of 19:00:03)})`
    - At L180, `oracleVersion == versionList[versionIndex]` i.e., `18:59:59 == 19:00:00` is false, so it won't enter `commitRequested()`
    - `commit()` does not have a check like `commitRequested()` L153-L157, so it can skip `versionIndex: 18`, even if `updateData` is a valid price for `versionIndex: 18`
- At this point, latest() returns OracleVersion(timestamp: 18:59:59, ...)
- `commitRequested({versionIndex: 18, updateData: (pyth data of 19:00:04)})`
    - `commitRequested()` does not have a check like `commit()` L199's newlatestVersion > oldlatestVersion (i.e., versionToCommit > _latestVersion)

- At this point, `latest()` returns OracleVersion(timestamp: 18:59:58, ...)

Recommendation

In addition to the invariant of `newPrice.publishTime > lastCommittedPublishTime`, make sure `newLatestVersion > oldLatestVersion`.

We've oped to fix this by setting the nextVersionIndexToCommit in the unrequested commit flow

fix: https://github.com/equilibria-xyz/perennial-v2/pull/100