Hyperdrive June 2024 Security Review

Auditors

Saw-Mon and Natalie, Lead Security Researcher

Report prepared by: Lucas Goiriz

June 29, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Hyperdrive is a new AMM protocol with a novel pricing mechanism for fixed and variable yield positions. It introduces terms on demand and removes the need for liquidity providers to roll over their capital allocations. Additionally, its mechanism design enables a more efficient, symmetrical yield market.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of hyperdrive according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 5 days in total, DELV engaged with Spearbit to review the hyperdrive protocol. In this period of time a total of **9** issues were found.

Summary

Project Name	DELV
Repository	hyperdrive
Commit	e91559...03c8
Type of Project	DeFi, Yield
Audit Timeline	Jun 10 to Jun 14
Two week fix period	Jun 17 - Jun 21

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	0	0	0
Low Risk	3	2	1
Gas Optimizations	1	1	0
Informational	4	1	3
Total	**9**	**5**	**4**

5 Findings

5.1 High Risk

5.1.1 `apr` circuit breaker can be avoided

Severity: High Risk

Context: HyperdriveBase.sol#L488-L496, HyperdriveLP.sol#L204-L214

Description: The check in `_addLiquidity` only compares the current spot `apr` to the `weightedSpotAPR` of the latest checkpoint:

```
// Enforce the slippage guard.
uint256 apr = HyperdriveMath.calculateSpotAPR(
    _effectiveShareReserves(),
    _marketState.bondReserves,
    _initialVaultSharePrice,
    _positionDuration,
    _timeStretch
);

// ...

// Perform a checkpoint.
uint256 latestCheckpoint = _latestCheckpoint();
_applyCheckpoint(
    latestCheckpoint,
    vaultSharePrice,
    LPMath.SHARE_PROCEEDS_MAX_ITERATIONS
);

// Ensure that the spot APR is close enough to the previous weighted
// spot price to fall within the tolerance.
{
    uint256 weightedSpotAPR = HyperdriveMath.calculateAPRFromPrice(
        _checkpoints[latestCheckpoint].weightedSpotPrice,
        _positionDuration
    );
    if (
        apr > weightedSpotAPR + _circuitBreakerDelta ||
        (weightedSpotAPR > _circuitBreakerDelta &&
            apr < weightedSpotAPR - _circuitBreakerDelta)
    ) {
        revert IHyperdrive.CircuitBreakerTriggered();
    }
}
```

1. Big price movements across checkpoint boundaries.

   But there could be big movements on the curve just before the current checkpoint start time, for example opening a max short. These price movements would not get included in the weighted spot price calculation of the next checkpoint and this:

   - Big price/apr/curve movement and...
   - `_addLiquidity`

   trades can happen with a short time difference (1 seconds or smallest delay between blocks) and circuit breaker logic can be avoided.

   For example, apply the following patch to `test/integrations/hyperdrive/SandwichTest.t.sol`:

```
 diff --git a/test/integrations/hyperdrive/SandwichTest.t.sol
↪  b/test/integrations/hyperdrive/SandwichTest.t.sol
 index 6c54eadf..78fcf38a 100644
 --- a/test/integrations/hyperdrive/SandwichTest.t.sol
 +++ b/test/integrations/hyperdrive/SandwichTest.t.sol
 @@ -385,6 +385,9 @@ contract SandwichTest is HyperdriveTest {
             2 *
             hyperdrive.getPoolConfig().minimumShareReserves;

 +        // Most of the term passes and no interest accrues.
 +        advanceTime(POSITION_DURATION - 1 seconds, 0);
 +
          // Celine opens a large short.
          shortAmount = shortAmount.normalizeToRange(
              hyperdrive.getPoolConfig().minimumTransactionAmount,
 @@ -392,6 +395,9 @@ contract SandwichTest is HyperdriveTest {
          );
          openShort(celine, shortAmount);

 +        // Rest of the term passes and no interest accrues.
 +        advanceTime(1 seconds, 0);
 +
          // Celine adds liquidity.
          vm.stopPrank();
          vm.startPrank(celine);
```

2. Price movements:
    1. Perform the max short (or almost the max amount) on the checkpoint start time.
    2. Wait a block
    3. Perform a tiny trade that would only move the point p on the curve C a little.
    4. Add liquidity's circuit breaker does not get triggered since the weighted average spot price would be the exact spot price right after the max short trade in step 1.

    See the issue regarding "`weightedSpotPrice`".

```
 diff --git a/test/integrations/hyperdrive/SandwichTest.t.sol
↪  b/test/integrations/hyperdrive/SandwichTest.t.sol
 index 6c54eadf..09724bd8 100644
 --- a/test/integrations/hyperdrive/SandwichTest.t.sol
 +++ b/test/integrations/hyperdrive/SandwichTest.t.sol
 @@ -387,11 +387,15 @@ contract SandwichTest is HyperdriveTest {

          // Celine opens a large short.
          shortAmount = shortAmount.normalizeToRange(
 -            hyperdrive.getPoolConfig().minimumTransactionAmount,
 -            hyperdrive.calculateMaxShort()
 +            2 * hyperdrive.getPoolConfig().minimumTransactionAmount,
 +            hyperdrive.calculateMaxShort() - hyperdrive.getPoolConfig().minimumTransactionAmount
          );
          openShort(celine, shortAmount);

 +        // let one block pass
 +        advanceTime(12 seconds, 0);
 +        openShort(celine, hyperdrive.getPoolConfig().minimumTransactionAmount);
 +
          // Celine adds liquidity.
          vm.stopPrank();
          vm.startPrank(celine);
```

Recommendation: One can avoid the above issues by using a moving window weighted average price or make sure some form of weighted average `apr` of the previous checkpoint is included in the circuit breaker.

Spearbit: PR 1058 does address the points mentioned in this issue, but it does not resolve the original issue completely. Let's assume the weighted average apr from the previous checkpoint is a_{prev} and assume during this checkpoint most of the trades happen around the max short point with a_{curr} so at the end when we advance to the next checkpoint the `apr` used in the circuit breaker would be a_{curr} then the user adds liquidity at this max short position and then opens the max long and thus performs the type of attacks we were trying to block. Although now it would kind of require a long-range attack and thus it might be less likely.

One way to fix it might be to instead of for each checkpoint only considering the average spot prices of that particular checkpoint, one can keep the average from the checkpoint in question and also a number of checkpoints before it. Again this suggestion would also need to be investigated.

Delv: If someone opens a large short and has to wait a block or two or three to complete the attack, then someone will likely come in and long the price back down and wreck their position. in this sense, they are taking a huge financial risk for very little reward.

In the other case where the fixed rate is correct (close to the variable rate), opening this large long is unprofitable since they are taking on a bunch of lower-than-market rate bonds, and the market will be shorted back to where it was profitably.

Spearbit: In general depending on how far back from the current timestamp the weighted average spot price is taken makes this category of attacks would be less and less likely. And so, one needs to consider how far back they think they should take the average and asses the risks and likelihood.

Marking this as verified and leaving the analysis to the client to assess for which pool parameters/config sets it would make sense to keep the average to just one checkpoint before the current one.

5.2 Low Risk

5.2.1 Weighted spot price is not updated for all curve movements

Severity: Low Risk

Context: HyperdriveLP.sol#L157, HyperdriveLP.sol#L326, HyperdriveLP.sol#L409

Description: In the following routes:

- `_addLiquidity`
- `_removeLiquidity`
- `_redeemWithdrawalShares`

weighted spot price does not get updated for the latest checkpoint. One might argue that thus might be because the point $P \in C$ on our curve only gets scaled and thus the spot price before and after the call to these routes stay the same. But this is not completely true, since due to algebraic imprecisions these before and after spot prices might be slightly different.

Recommendation: To take into consideration the differences in spot prices before and after the call to these endpoints, make sure to updated the weighted spot price for the current checkpoint using the spot price before the pool state gets updated in these routes.

> Note that the similar update for closing matured longs or shorts is not required and is not implemented since the following is performed before hand (HyperdriveLong.sol#L186-L191, HyperdriveShort.sol#L205-L210):

```
_applyCheckpoint(
    _maturityTime,
    vaultSharePrice,
    LPMath.SHARE_PROCEEDS_MAX_ITERATIONS,
    true
);
```

and the last price for a passed checkpoint can be only included once.

Delv: Client decided not to fix this. The discrepancies between spot prices (if there are any) are so minuscule that we are not concerned about this issue

Spearbit: Acknowledged.

5.2.2 Non-safe operations used in safe routes

Severity: Low Risk

Context: FixedPointMath.sol#L382-L384, LPMath.sol#L1001, LPMath.sol#L1071

Description: In `calculateDistributeExcessIdleShareProceeds` we have the following code blocks:

```
if (smallestDelta == 0 || delta.abs() < smallestDelta.abs()) {
    smallestDelta = delta;
    closestShareProceeds = shareProceeds;
    closestPresentValue = presentValue;
}
// ...
if (lastDelta.abs() < smallestDelta.abs()) {
    closestShareProceeds = shareProceeds;
    closestPresentValue = presentValue_;
}
```

and the implementation of `abs` is:

```
function abs(int256 a) internal pure returns (int256) {
    return a < 0 ? -a : a;
}
```

and thus note that we have:

```
abs(type(int256).min) // this would revert
```

Recommendation: One can used modified version of `abs` where `abs(type(int256).min)` is defined as `abs(type(int256).max)` and thus the overflow is avoided by clamping.

Delv: PR 1062 fixes the issue by using the `abs` from OpenZeppelin `SignedMath` library which basically uses the same implementation but in an `unchecked` block:

```
function abs(int256 n) internal pure returns (uint256) {
    unchecked {
        // must be unchecked in order to support `n = type(int256).min`
        return uint256(n >= 0 ? n : -n);
    }
}
```

Spearbit: Verified.

5.2.3 `returndata` bombing by `_checkpointRewarder`

Severity: Low Risk

Context: HyperdriveCheckpoint.sol#L335-L340

Description: In `_applyCheckpoint` the following `call` is being made if there exists a `_checkpointRewarder`:

```
if (_checkpointRewarder != address(0)) {
    bool isTrader = _isTrader; // avoid stack-too-deep
    (bool _success, ) = _checkpointRewarder.call(
        abi.encodeCall(
            IHyperdriveCheckpointRewarder.claimCheckpointReward,
            (msg.sender, checkpointTime, isTrader)
        )
    );
    // NOTE: Avoid unused local variable warning.
    _success;
}
```

Using the above pattern `solc` compiler copies the return data to memory even though the return data is not being used.and thus a potentially malicious or misconfigured `_checkpointRewarder` can return a very large return data that would cause the above block to run out of gas.

Recommendation: Use low-level assembly to call the `_checkpointRewarder` and check the `success` status. One can use already made libraries such as ExcessivelySafeCall, and since the returned data from the call to `_checkpointRewarder` is not used one can set the `maxCopy` to `0`:

```
if (_checkpointRewarder != address(0)) {
    bool isTrader = _isTrader; // avoid stack-too-deep
    _checkpointRewarder.excessivelySafeCall(
        gasleft(),
        0, // value of 0
        0, // don't copy returndata to memory
        abi.encodeCall(
            IHyperdriveCheckpointRewarder.claimCheckpointReward,
            (msg.sender, checkpointTime, isTrader)
        )
    );
}
```

and to be even more gas efficient, one can inline the assembly to avoid all the extra opcodes related to the returned `success` and the `maxCopy` checks, etc...

```
if (_checkpointRewarder != address(0)) {
    bool isTrader = _isTrader; // avoid stack-too-deep
    bytes memory _calldata = abi.encodeCall(
        IHyperdriveCheckpointRewarder.claimCheckpointReward,
        (msg.sender, checkpointTime, isTrader)
    );

    assembly {
        pop(
            call(
                gas(), // gas
                sload(_checkpointRewarder.slot), // recipient
                0, // ether value
                add(_calldata, 0x20), // inloc
                mload(_calldata), // inlen
                0, // outloc
                0 // outlen
            )
        )
    }
}
```

Delv: Fixed in PR 1050 and PR 1072 using the 1st recommendation.

Spearbit: Verified.

5.3 Gas Optimization

5.3.1 `_effectiveShareReserves()` and `_marketState.bondReserves` can be cached in `_calculate(Open|Close)(Long|Short)`

Severity: Gas Optimization

Context: HyperdriveLong.sol#L454-L455, HyperdriveLong.sol#L465-L466, HyperdriveLong.sol#L586-L587, HyperdriveLong.sol#L602-L603, HyperdriveShort.sol#L477-L478, HyperdriveShort.sol#L498-L499, HyperdriveShort.sol#L598-L599, HyperdriveShort.sol#L610-L611

Description: In the above context in the `_calculate(Open|Close)(Long|Short)` (`_f`) internal functions we have:

```
function _f(/*...*/) /*...*/ {
   // ...
   (/*...*/) = HyperdriveMath.f(
      _effectiveShareReserves(),
      _marketState.bondReserves,
      // ...
   );
   // ...
   spotPrice = HyperdriveMath.calculateSpotPrice(
       _effectiveShareReserves(),
       _marketState.bondReserves,
       // ...
   );
   // ...
}
```

and thus storage slots are read multiple times.

Recommendation: It would be best to cache `_effectiveShareReserves()` and `_marketState.bondReserves` to avoid multiple storage reads.

Delv: Fixed in PR 1060.

Spearbit: Verified.

5.4 Informational

5.4.1 Picking different initial point to address K inflation should be analyzed more throughly

Severity: Informational

Context: LPMath.sol#L76-L126

Description: Both before and after the code change for picking the initial point $(z-\zeta, y) \in C$, the following invariant is satisfied which means both of these points stay on the same ray passing through the origin which sets the given target price or `apr` and thus the different approach in initialisation is kind of just scaling the curve. The invariant is:

$$p_{target} = \frac{1}{1 + a \cdot \Delta t_{pos}} = \frac{\mu(z-\zeta)}{y}$$

$^{t_s} = p_{spot}$

The general form of how this initialisation point is set is given by the extra constraint that (for a given scaling parameter λ):

$$x = cz = c(z-\zeta) + p_{spot} \cdot y \cdot \lambda$$

which gives us:

$$y = \frac{z}{\frac{p_{spot}}{c}\lambda + \frac{p_{spot}^{1/t_s}}{\mu}}$$

and

$$\zeta = \frac{p_{spot} \cdot y}{c}\lambda$$

and so λ let's us interpolate between the 2 different implementations

1. Case $\lambda = 0$:

   This is the case before the code change which gives us:

   $$y = \mu(1 + a\Delta t_{pos})^{1/t_s} z$$

   and $\zeta = 0$

2. Case $\lambda = 1$:

   This is the new implementation where we have:

   $$y = \frac{z}{\frac{p_{spot}}{c} + \frac{p_{spot}^{1/t_s}}{\mu}}$$

   and

   $$\zeta = \frac{p_{spot} \cdot y}{c}$$

In both cases:

- z still represent the initial share of the `HyperDrive` pool in the vault.
- ζ, y are picked so that the spot price of the point $(z - \zeta, y)$ on the curve C (p_{spot}) is the target price p_{target} for the given `apr` a.

  Note that:

  - z_{min} is kept the same between the old and new implementation of the starting point which limits how far the `apr` can be pushed when one opens a max short then immediately adds liquidity.
  - The most important part is that upon initialisation the values of ζ and y are comparable:

$$\zeta = \frac{p_{spot} \cdot y}{c}\lambda$$

This fact is pretty important and has 2 consequences:

1. When opening a max short and then adding liquidity the scale factor w is given by:

$$w = \frac{z_{min} + \zeta + \Delta z}{z_{min} + \zeta}$$

and thus w cannot blow up when λ is around or greater than 1. and note that how w can blowup when ζ is for example 0:

$$w_{\zeta=0} = \frac{z_{min} + \Delta z}{z_{min}}$$

2. The solvency line also doesn't blow up. The solvency line after the max short and add liquidity trade is given by (below z is just a parameter to define the equation):

$$L_{solve}(z) = -c(z + w\zeta - z_{min}) + w \cdot y_{min} + L^{old}_{exp}$$

Here is how to derive this equation accurately:

Assume we are at the point $P_0 = (z_0 - \zeta_0, y_0) \in C_0$ before adding liquidity and assume adding liquidity scales the curve by w and so we end up at $P_1 = (z_1 - \zeta_1, y_1) = (wz_0 - w\zeta_0, wy_0) \in wC_0 = C_1$.

The solvency criteria at point P_0 is:

$$z_0 \geq \frac{L_{exp,0}}{c_0} + z_{min}$$

also note that:

$$L_{exp,0} = \sum_{t_m} \max(0, s_{L,t_m} - s_{S,t_m})$$

and assume t_{m_0} is the maturity time for any trade that happens at the current checkpoint then:

$$L_{exp,0} = \sum_{t_m \neq t_{m_0}} \max(0, s_{L,t_m} - s_{S,t_m})$$

$+ \max(0, s_{L,t_{m_0}} - s_{S,t_{m_0}}) = L'_{exp,0} + \max(0, s_{L,t_{m_0}} - s_{S,t_{m_0}})$

and so the solvency inequality at point P_0 can be split into 2 inequalities so we can get rid of the last use of max:

1. This inequality enforces the solvency for all checkpoints except t_{m_0} and happens when $s_{L,t_{m_0}} - s_{S,t_{m_0}}$ is negative:

$$z_0 \geq \frac{L'_{exp,0}}{c_0} + z_{min}$$

2.

$$z_0 \geq \frac{L'_{exp,0}}{c_0} + \frac{s_{L,t_{m_0}} - s_{S,t_{m_0}}}{c_0} + z_{min}$$

We are going to focus on the second inequality to define a solvency line L_{P_0}. Assume the next trade (opening or closing a short or long) moves our point P_0 by $(\Delta z, \Delta y)$ where:

$$P'_0 = (z, y) = P_0 + (\Delta z, \Delta y) = (z_0 - \zeta_0 + \Delta z, y_0 + \Delta y)$$

In the above (z, y) represent coordinates in the (z_e, y) domain using the language by `Delv`.

Then the solvency requirement according to the inequality 2. would be:

$$z_0 + \Delta z \geq \frac{L'_{exp,0}}{c_0} + \frac{s_{L,t_{m_0}} - s_{S,t_{m_0}} - \Delta y}{c_0} + z_{min}$$

or

$$z + \zeta_0 \geq \frac{L'_{exp,0}}{c_0} + \frac{s_{L,t_{m_0}} - s_{S,t_{m_0}} + y_0 - y}{c_0} + z_{min}$$

or

$$y \geq -c_0(z + \zeta_0 - z_{min}) + (L'_{exp,0} + s_{L,t_{m_0}} - s_{S,t_{m_0}} + y_0)$$

and thus a solvency line at point P_0 is defined as:

$$L_{P_0} : y = -c_0(z + \zeta_0 - z_{min}) + (L'_{exp,0} + s_{L,t_{m_0}} - s_{S,t_{m_0}} + y_0)$$

and we can only move on the curve C_0 above this line. Now when we add liquidity and scale our point P_0 to $P_1 = wP_0$ to end up on the curve $C_1 = wC_0$, the long exposure and it's related parameters do not change and we can follow the above steps to derive a solvency line at the point P_1:

$$L_{P_1}(z) : y = -c_1(z + \zeta_1 - z_{min}) + (L'_{exp,0} + s_{L,t_{m_0}} - s_{S,t_{m_0}} + y_1)$$

or:

$$L_{P_1}(z) : y = -c_1(z + w\zeta_0 - z_{min}) + (L'_{exp,0} + s_{L,t_{m_0}} - s_{S,t_{m_0}} + wy_0)$$

or

$$L_{P_1}(z) = L_{P_0}(z) - (c_1 - c_0)(z - z_{min}) - (c_1 w - c_0)\zeta_0 + (w - 1)y_0$$

and so if $c_1 \approx c_0$ and ζ_0 is not comparable to y_0 and w is big enough one can really translate up the solvency line so that it would hit the new curve C_1 at a much higher `apr` rate and thus lock the pool in those high interest rate regions.

parameter	description
y_{min}	is the amount of bond reserves after the max short trade when the effective share reserves is basically around z_m
L_{exp}^{old}	the long exposure after the max short trade

Note that if ζ was pretty small or close to 0 compared to y_{min} how the equation of the L_{solve} could blow it up so that it would interest the scaled curve wC very close to the point where one had ended up after adding liquidity. Thus would mean one could only trade on a small portion of the scaled curve to keep the pool solvent and thus the `apr` would be locked in a high region and one would not be able to bring it back to 0 or close to its initial `apr`. Here is a graph to view this issue.

Now the question is since ζ is basically the sum:

- It's initial value.
- Any added zombie interests.
- Flat trades.

and it also scales when one adds or removes liquidity or when there is a negative interest.

Then the above blowing up the L_{solve} line might be possible again if ζ ends up being a very small or even negative value.

3. Studying K

we have:

$$K_{\lambda=0} = (\mu z)^{1-t_s} \left(\frac{c}{\mu} + p_{spot}^{\frac{-(1-t)}{t}} \right.$$

and

$$K_{\lambda=1} = (\mu z)^{1-t_s} \left(\frac{c}{\mu} 1 - \frac{p_{spot}/c}{\frac{p_{spot}}{c} + \frac{p_{spot}^{1/t_s}}{\mu}} \right.$$

$$^{1-t} + \frac{1/\mu}{\frac{p_{spot}}{c} + \frac{p_{spot}^{1/t_s}}{\mu}}^{1-t}$$

It is obvious that $K_{\lambda=1} < K_{\lambda=0}$ and thus the curve has been scaled down and the initial point has been moved closer to the origin on the desired `apr` ray.

Recommendation: Underlying meaning of ζ, y should be documented and the picking different initial point to address `K` inflation should be analyzed more thoroughly.

Delv: Acknowledged.

Spearbit: Acknowledged.

5.4.2 Different ways to calculate weighted average `apr`

Severity: Informational

Context: HyperdriveBase.sol#L488-L496, HyperdriveLP.sol#L204-L214

Description: We have:

$$a = \frac{1}{\Delta t_{pos}} \frac{1}{p} - 1$$

= f(p)

parameter	description
a	spot `apr`
Δt_{pos}	position duration
p	spot price

Note that f is a convex function and thus for weights $w_i \geq 0$ where $\sum w_i = 1$ we have:

$$f(\sum w_i \cdot p_i) \leq \sum w_i \cdot f(p_i)$$

The weighted average `apr` (a_{av}) is taken to be $f(\sum w_i \cdot p_i)$.

Recommendation: It should be documented why $\sum w_i \cdot f(p_i)$ is not used for a_{av}. This could have been it because to favour anchoring towards smaller `aprs` (that can kind of make sense where the high `apr` regions are not so desired).

Delv: Acknowledged.

Spearbit: Acknowledged.

5.4.3 `weightedSpotPrice`

Severity: Informational

Context: `weightedSpotPrice`

Description: `weightedSpotPrice` is basically calculated per checkpoint as (up to division errors):

1. Actively minted checkpoints, for checkpoints that have been minted when they were active

$$p_{av} = \frac{\Delta t_{in} \cdot p_{in} + (\sum \Delta t_i \cdot p_i) + \Delta t_f \cdot p_f}{\Delta t_{in} + (\sum \Delta t_i) + \Delta t_f}$$

where p_{in} is the spot price for before the first trade in that checkpoint when the checkpoint gets minted and p_f is comes from minting the next checkpoint.

Note that the summation above is over all trades that happened during that checkpoint including:

endpoint	checkpoint	contribute p_{in} only	non-matured only
`_openLong`	latest checkpoint		
`_closeLong`	latest checkpoint		✓

endpoint	checkpoint	contribute p_{in} only	non-matured only
`_openShort`	latest checkpoint		
`_closeShort`	latest checkpoint		✓
`_checkpoint`	-	✓	
`_initialize`	latest checkpoint	✓	
`_addLiquidity`	latest checkpoint	✓	
`_removeLiquidity`	latest checkpoint	✓	
`_redeemWithdrawalShares`	latest checkpoint	✓	

and all the p_i spot prices are the the prices calculated on the curve C before the trade / before the point p being moved.

p_f is either:

- The spot price right after the last trade in the checkpoint which would be the first spot price before the first trade in the next checkpoint or...
- Is the weighted average spot price for the next non-delayed minted checkpoint.

2. Delayed minted checkpoints, for checkpoint that get minted due to a delay

$$p_{av} = p_{av,next}$$

where $p_{av,next}$ is the weighted average spot price of the next minted checkpoint.

Delv: Acknowledged.

Spearbit: Acknowledged.

5.4.4 Calculating `checkpointVaultSharePrice` and `checkpointWeightedSpotPrice` can be simplified

Severity: Informational

Context: HyperdriveCheckpoint.sol#L90-L140

Description: In `_applyCheckpoint` we have:

```
// If the checkpoint time is the latest checkpoint, we use the current
// vault share price and spot price. Otherwise, we use a linear search
// to find the closest non-zero vault share price and use that to
// perform the checkpoint. We use the weighted spot price from the
// checkpoint with the closest vault share price to populate the
// weighted spot price.
uint256 checkpointVaultSharePrice;
uint256 checkpointWeightedSpotPrice;
uint256 latestCheckpoint = _latestCheckpoint();
if (_checkpointTime == latestCheckpoint) {
    checkpointVaultSharePrice = _vaultSharePrice;
    checkpointWeightedSpotPrice = HyperdriveMath.calculateSpotPrice(
        _effectiveShareReserves(),
        _marketState.bondReserves,
        _initialVaultSharePrice,
        _timeStretch
    );
} else {
    for (
```

```
        uint256 time = _checkpointTime + _checkpointDuration;
        ;
        time += _checkpointDuration
    ) {
        // If the time is the latest checkpoint, we use the vault share
        // price and the current spot price.
        if (time == latestCheckpoint) {
            checkpointVaultSharePrice = _vaultSharePrice;
            checkpointWeightedSpotPrice = HyperdriveMath
                .calculateSpotPrice(
                    _effectiveShareReserves(),
                    _marketState.bondReserves,
                    _initialVaultSharePrice,
                    _timeStretch
                );
            break;
        }

        // If the time isn't the latest checkpoint, we check to see if
        // the checkpoint's vault share price is non-zero. If it is,
        // that is the vault share price that we'll use to create the
        // new checkpoint. We'll use the corresponding weighted spot
        // price to instantiate the weighted spot price for the new
        // checkpoint.
        checkpointVaultSharePrice = _checkpoints[time].vaultSharePrice;
        if (checkpointVaultSharePrice != 0) {
            checkpointWeightedSpotPrice = _checkpoints[time]
                .weightedSpotPrice;
            break;
        }
    }
}
```

The above code has an `if` / `else` branch as well as a `for` loop in one of the branches. It also includes 2 `break` statements where one had forgotten before one of the PRs and had caused an infinite loop in some cases.

Recommendation: For readability and also avoiding errors it might be better to simplify the above pattern. And also note how there are parts that can be refactored. Thus one can implement the same calculation in one of the two ways below the first one being slightly more gas costly compared to the current implementation:

1. First way:

```
// By default, we use the current
// vault share price and spot price. Otherwise, we use a linear search
// to find the closest non-zero vault share price and use that to
// perform the checkpoint. We use the weighted spot price from the
// checkpoint with the closest vault share price to populate the
// weighted spot price.
uint256 checkpointVaultSharePrice = _vaultSharePrice;
uint256 checkpointWeightedSpotPrice = HyperdriveMath.calculateSpotPrice(
    _effectiveShareReserves(),
    _marketState.bondReserves,
    _initialVaultSharePrice,
    _timeStretch
);

uint256 latestCheckpoint = _latestCheckpoint();
uint256 nextCheckpointTime = _checkpointTime + _checkpointDuration;

for (; nextCheckpointTime < latestCheckpoint;) {
    // If the time isn't the latest checkpoint, we check to see if
    // the checkpoint's vault share price is non-zero. If it is,
    // that is the vault share price that we'll use to create the
    // new checkpoint. We'll use the corresponding weighted spot
    // price to instantiate the weighted spot price for the new
    // checkpoint.
    uint256 price = _checkpoints[nextCheckpointTime].vaultSharePrice;
    if (price != 0) {
        checkpointVaultSharePrice = price;
        checkpointWeightedSpotPrice = _checkpoints[nextCheckpointTime]
            .weightedSpotPrice;
        break;
    }

    nextCheckpointTime += _checkpointDuration;
}
```

2. Or to accommodate for the past checkpoints:

```
uint256 checkpointVaultSharePrice;
uint256 checkpointWeightedSpotPrice;

uint256 latestCheckpoint = _latestCheckpoint();
uint256 nextCheckpointTime = _checkpointTime + _checkpointDuration;

for (; nextCheckpointTime < latestCheckpoint;) {
    // If the time isn't the latest checkpoint, we check to see if
    // the checkpoint's vault share price is non-zero. If it is,
    // that is the vault share price that we'll use to create the
    // new checkpoint. We'll use the corresponding weighted spot
    // price to instantiate the weighted spot price for the new
    // checkpoint.
    uint256 price = _checkpoints[nextCheckpointTime].vaultSharePrice;
    if (price != 0) {
        checkpointVaultSharePrice = price;
        checkpointWeightedSpotPrice = _checkpoints[nextCheckpointTime]
            .weightedSpotPrice;
        break;
    }

    nextCheckpointTime += _checkpointDuration;
}

if (checkpointVaultSharePrice == 0) {
    checkpointVaultSharePrice = _vaultSharePrice;
    checkpointWeightedSpotPrice = HyperdriveMath.calculateSpotPrice(
        _effectiveShareReserves(),
        _marketState.bondReserves,
        _initialVaultSharePrice,
        _timeStretch
    );
}
```

Delv: Fixed in PR 1059 below using the 2nd suggestion plus some renaming and optimizations.

Spearbit: Verified.