SECURITY REVIEW

CENTRIFUGE V3

PROTECTED BY RECON

GETRECON.XYZ
@GALLODASBALLO
@NICANOR

Recon Security Review

Introduction

Alex The Entreprenerd performed a 3 weeks Security Review of Centrifuge

Repos: https://github.com/centrifuge/protocol-v3

This review uses Code4rena Severity Classification

The Manual Review is done as a best effort service, while a lot of time and attention was dedicated to the security review, it cannot guarantee that no bug is left

As a general rule we always recommend doing one additional security review until no bugs are found, this in conjunction with a Guarded Launch and a Bug Bounty can help further reduce the likelihood that any specific bug was missed

Given the extensive amount of changes that have happened during the review we recommend that another review is done with a different team after the codebase has been finalized

Following that we recommend a Guarded Launch secured by a Bug Bounty and a Security Contest

These suggested next steps seem to be consistent with what the Centrifuge team has planned

We also wrote an extensive "Suggested Next Steps" to discuss in detail

Lastly we wrote "An auditor introduction to the Hub Codebase and its relation to Vaults" to help onboard new reviewers

Personally I believe that a diagram of intended vs unintended flows, would massively help newer reviewers, as it's quite difficult to discern which flows are "intended" vs which flows would be considered "unintended"

About Recon

Recon offers boutique security reviews, invariant testing development and is pioneering Cloud Fuzzing as a best practice by offering Recon Pro, the most complete tool to run tools such as Echidna, Medusa, Foundry, Kontrol and Halmos in the cloud with just a few clicks

About Alex

Alex is a well known Security Researcher that has collaborated with multiple contest firms such as:

- Code4rena - One of the most prolific and respected judges, won the Tapioca contest, at the time the 3rd highest contest pot ever
- Spearbit - Have done reviews for Tapioca, Threshold USD, Velodrome and more

- Recon - Centrifuge Invariant Testing Suite, Corn and Badger invariants as well as live monitoring

Additional Services by Recon

Recon offers:

- Invariant Testing Audits - We'll write your invariant tests then perform and audit on the code
- Cloud Fuzzing as a Service - The easiest way to run invariant tests in the cloud - Ask about Recon Pro
- Audits - High Quality Audits done by Highly Qualified Reviewers that work with Alex personally

Table of Contents

- Q-15 `Hub.updateHoldingAmount` introduces a lot of complexity but can only be used in scenarios that can be executed with simpler functions
  - Q-16 Queueing can be sidestepped with multiple accounts
  - Q-17 `userOrder.pending` update in `_updatePending` can be simplified
  - Q-18 Multiple `requestDeposit` and one `cancelDeposit` can result in undefined behaviour
  - Q-19 QA Findings - SCM
  - Q-20 `updateHoldingValue / updateJournal` can cause socialized losses if the admin doesn't `updateHoldingValuation` when shares value rapidly changes
  - Q-21 Gas Optimizations - SCM
  - Q-22 Permit wont' work with DAI
  - Q-23 It's generally best not to grant permissioneless execution
  - Q-24 `receiveWormholeMessages` is `payable` but the value is unused
  - Q-25 `Permissioneless Methods` lists `createPool` which has the modifier `auth`
  - Q-26 `_generateJournalId` can be changed to make full use of it's type
  - Q-27 `Holdings.update` can revert when the absolute difference in value is above `type(int128).max`
  - Q-28 `previewMint` rounding direction is incorrect
  - Q-29 `Vaults` `try/catch` may be subject to Low Gas Attack
- **Analysis**
  - A-01 Accounting Gotchas wrt processing yield, pending deposits and withdrawals
  - A-02 2 aspects to Pricing Shares when dealing with multi currencies
  - A-03 Collection of CEI non conforming code when including malicious hooks
  - A-04 `ShareClassManager.claimDepositUntilEpoch` can open up Race Condition in `claimXUntilEpoch` can create ghost `pendingX` due to calling `_postClaimUpdateQueued` when epochs are still being proceesed
  - A-05 Suggested Next Steps
  - A-06 An auditor introduction to the Hub Codebase and its relation to Vaults
- **Economic**
  - E-01 Sync Deposit / Mint can open up to risk free arbitrage when a positive price update is known
- **Invariants**
  - I-01 New Property Checks for Stateful Fuzzing

H-01 Casting of negative `int128` to `uint128` overflows, breaking accounting in `updateHolding`

Impact

The code for `updateHolding` looks as follows:

https://github.com/centrifuge/protocol-v3/blob/95b10b57af336a111ac9f82c25caf5dc0b0310e6/src/hub/Hub.sol#L328-L356

```
    function updateHolding(ShareClassId scId, AssetId assetId) public payable {
        _protectedAndUnlocked();

        int128 diff = holdings.update(unlockedPoolId, scId, assetId); /// @audit overflow case below is handled differently

        if (diff > 0) {
            if (holdings.isLiability(unlockedPoolId, scId, assetId)) {
                accounting.addCredit(
                    holdings.accountId(unlockedPoolId, scId, assetId, uint8(AccountType.Liability)), uint128(diff)
                );
                accounting.addDebit(
                    holdings.accountId(unlockedPoolId, scId, assetId, uint8(AccountType.Expense)), uint128(diff)
                );
            } else {
                accounting.addCredit(
                    holdings.accountId(unlockedPoolId, scId, assetId, uint8(AccountType.Gain)), uint128(diff)
                );
                accounting.addDebit(
                    holdings.accountId(unlockedPoolId, scId, assetId, uint8(AccountType.Asset)), uint128(diff)
                );
            }
        } else if (diff < 0) {
            if (holdings.isLiability(unlockedPoolId, scId, assetId)) {
                accounting.addCredit(
                    holdings.accountId(unlockedPoolId, scId, assetId, uint8(AccountType.Expense)), uint128(diff)
                );
                accounting.addDebit(
                    holdings.accountId(unlockedPoolId, scId, assetId, uint8(AccountType.Liability)), uint128(diff)
                );
```

It will call `holdings.update` which can return a positive or negative value

For a positive value, casting is safe as the `uint128` is bigger than `int128.max`

For a negative value, the casting is unsafe as the `negative flag` from the `int128` will cause the compiler to interpret the value as one of the highest possible `uint128` (the ones above `int128.max`)

This has a dramatic impact on the code, breaking accounting

POC

```
[PASS] test_overflow() (gas: 6645)
Logs:
  x 123
  converted_x 123
  y -123
  converted_y 340282366920938463463374607431768211333
```

```
  // SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

import {FoundryAsserts} from "@chimera/FoundryAsserts.sol";

import "forge-std/console2.sol";

import {Test} from "forge-std/Test.sol";
import {TargetFunctions} from "./TargetFunctions.sol";


// forge test --match-contract CryticToFoundry -vv
contract CryticToFoundry is Test, TargetFunctions, FoundryAsserts {
    function setUp() public {
        setup();

        targetContract(address(counter));
    }

    // forge test --match-test test_crytic -vvv
    function test_crytic() public {
        // TODO: add failing property tests here for debugging
    }

    function test_overflow() public {
        int128 x = 123;
        int128 y = -123;

        uint128 converted_x = uint128(x);
        uint128 converted_y = uint128(y);

        console2.log("x", converted_x);
        console2.log("converted_x", converted_x);

        console2.log("y", y);
        console2.log("converted_y", converted_y);

    }
}
```

Mitigation

Change the code to subtract the correct absolute value

From experience using `int` is error prone, you should consider using `uint` with a `bool isNegative` Or alternatively harden the code by extensively reviewing these castings

With the expectation that any casting that is not fuzzed / FVd is probably unsafe

I believe the code change would pass `uint128(uint256(-int256(diff)))` which would work for all values including the most negative value of `int128`

M-01 `AsyncRequests._withdraw` calls `balanceSheet.withdraw` which triggers `sender.sendUpdateHoldingAmount` but the value from the `withdrawal` should have been triggered when `revokeShares` was called

Impact

`AsyncRequests .withdraw` updates the balance sheet and by consequence the hub, when it should instead be updated when `revokeShares` is called, otherwise the shares and underlying assets will report an incorrect change in valuation that stems from the discrepancy between:

- The price the user receives
- The price the Vault reports
- The real price the Hub should use

https://github.com/centrifuge/protocol-v3/blob/c50ee4c37680d6b15951f998898dea8998c06973/src/vaults/AsyncRequests.sol#L453-L469

```
    function _withdraw(address vaultAddr, address receiver, uint128 assets) internal {
        VaultDetails memory vaultDetails = poolManager.vaultDetails(vaultAddr);

        IAsyncVault vault_ = IAsyncVault(vaultAddr);
        PoolId poolId = vault_.poolId();
        ShareClassId scId = vault_.scId();
        /// @audit should this check validity? Wouldn't this cause issues to the user?
        (D18 pricePoolPerAsset,) = poolManager.pricePoolPerAsset(poolId, scId,
vaultDetails.assetId, true);
        IPoolEscrow(address(poolEscrowProvider.escrow(poolId))).reserveDecrease(
            scId, vaultDetails.asset, vaultDetails.tokenId, assets
        );

        balanceSheet.withdraw(
            poolId, scId, vaultDetails.asset, vaultDetails.tokenId, receiver, assets,
pricePoolPerAsset
        );
    }
```

`pricePoolPerAsset` is highly likely to be different than the one that the user will have (`maxWithdraw`, `redeemPrice`)

`BalanceSheet.withdraw` will call `sendUpdateHoldingAmount`

https://github.com/centrifuge/protocol-v3/blob/c50ee4c37680d6b15951f998898dea8998c06973/src/vaults/BalanceSheet.sol#L236-L238

```
        emit Withdraw(poolId, scId, asset, tokenId, receiver, amount, pricePoolPerAsset,
uint64(block.timestamp));
        sender.sendUpdateHoldingAmount(poolId, scId, assetId, receiver, amount, pricePoolPerAsset,
false);
    } /// @audit Looks wrong
```

This is subtracting the holding value at the current time, with the current price

Whereas in the previous version the value change would have been recorded in `revokeShares`

Pseudocode of old version

```

    function revokeShares(D18 navPerShare)
        public
    {
        (uint128 payoutAssetAmount,) = shareClassManager.revokeShares(poolId, scId, payoutAssetId,
navPerShare, valuation);
        uint128 valueChange = holdings.decrease(poolId, scId, payoutAssetId, valuation,
payoutAssetAmount);

        // TODO: Overflow Properties here as well
        lte(payoutAssetAmount, depositAmt, "payoutAssetAmount > depositAmt");
        lte(valueChange, depositValue, "valueChange > depositValue");

        depositAmt -= payoutAssetAmount;
        depositValue -= valueChange;


        _unlock();
        accounting.addCredit(
            ASSET_ACCOUNT, valueChange
        );
        accounting.addDebit(
            EQUITY_ACCOUNT, valueChange
        );
        _lock();
```

The implications of this are the following:

- If you remove the assets when the shares are revoked, the system will not be subject to more value fluctuations that can happen between when the revoke has happened (time at which the withdrawal price is computed) and when the assets are withdrawn by the user
- If you keep the code as is, then the `_withdraw` operation is subject to being priced in 3 different ways: The user price (price, maxDeposit), the `pricePoolPerAsset` when the user calls `_withdraw`, the real price of the `payoutAssetId` that will be revealed only when the admin calls `updateHoldingValue`

Mitigation - TODO

I believe the change in value should happen on `revokeShares` or after `BalanceSheet.revokedShares` is called https://github.com/centrifuge/protocol-v3/blob/4a51fd1d25891cce91e2dbcbc5443b5024701b54/src/vaults/BalanceSheet.sol#L192-L198

```
    /// @inheritdoc IBalanceSheetGatewayHandler
    function revokedShares(PoolId poolId, ShareClassId scId, AssetId assetId, uint128 assetAmount) external auth {
        (address asset, uint256 tokenId) = poolManager.idToAsset(assetId);
        /// @audit should update the amount here, or the amount should have been updated by the call triggering this
        // Lock assets to ensure they are not withdrawn and are available for the redeeming user
        poolEscrowProvider.escrow(poolId).reserveIncrease(scId, asset, tokenId, assetAmount);
    }
```

M-02 `triggerRedeemRequest` with non zero `tokensToTransfer` can be evaded by transferring to another user

https://github.com/centrifuge/protocol-v3/blob/c50ee4c37680d6b15951f998898dea8998c06973/src/vaults/AsyncRequests.sol#L306-L342

```
function triggerRedeemRequest(PoolId poolId, ShareClassId scId, address user, AssetId assetId,
uint128 shares)
        public
        auth
    {
        require(shares != 0, ShareTokenAmountIsZero());
        address vault_ = vault[poolId][scId][assetId];

        // If there's any unclaimed deposits, claim those first
        AsyncInvestmentState storage state = investments[vault_][user];
        uint128 tokensToTransfer = shares;
        if (state.maxMint >= shares) {
            // The full redeem request is covered by the claimable amount
            tokensToTransfer = 0;
            state.maxMint = state.maxMint - shares;
        } else if (state.maxMint != 0) {
            // The redeem request is only partially covered by the claimable amount
            tokensToTransfer = shares - state.maxMint;
            state.maxMint = 0;
        }

        require(_processRedeemRequest(vault_, shares, user, msg.sender, true),
FailedRedeemRequest());

        // Transfer the token token amount that was not covered by tokens still in escrow for
claims,
        // from user to escrow (lock share class tokens in escrow)
        if (tokensToTransfer != 0) {
            require(
                IShareToken(address(IAsyncVault(vault_).share())).authTransferFrom(
                    user, user, address(poolEscrowProvider.escrow(poolId)), tokensToTransfer
                ),
                ShareTokenTransferFailed()
            );
        }

        (address asset, uint256 tokenId) = poolManager.idToAsset(assetId);
        emit TriggerRedeemRequest(poolId.raw(), scId.raw(), user, asset, tokenId, shares);
        IAsyncVault(vault_).onRedeemRequest(user, user, shares);
    }
```

M-03 `Account.accountValue` can overflow in a few scenarios

Paste into any Foundry Test

The accounting logic seems to be susceptible to these edge cases:

- Overflow Caught by Compiler Revert when `int128` values go past their range
- Silent overflow when casting `uint128` to values past `int128`

Logs:

```
Encountered 1 failing test in test/Counter.t.sol:CounterTest
[FAIL: panic: arithmetic underflow or overflow (0x11); counterexample:
calldata=0x88db638d000000000000000000000000000000000000000000000000000000000000000100000000000000000
00000000000000000000000000000000000000000000aa5000000000000000000000000000000008000000000000000000
00000000000 args=[true, 2725, 170141183460469231731687303715884105728 [1.701e38]]]
test_revert_overflows(bool,uint128,uint128) (runs: 46, μ: 1126, ~: 1131)

[FAIL: Doesn't loop around: -170141183460469231731687303715884105728 <= 0] test_overflow_normal()
(gas: 4305)
```

Code:

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";

contract CounterTest is Test {

    function setUp() public {
    }

    // It can revert when it's too low
    function test_revert_overflows(bool isDebitNormal, uint128 totalDebit, uint128 totalCredit) public {
        /// @audit Revert 1: If `totalDebit - totalCredit` < type(int128).min then it will revert
        int128 delta = accountValue(isDebitNormal, totalDebit, totalCredit);
    }

    // It can loop around when it's too high
    function test_overflow_normal() public {
        bool isDebitNormal = true;
        uint128 totalDebit = uint128(type(int128).max) + 1;
        uint128 totalCredit = 0;
        int128 val = accountValue(isDebitNormal, totalDebit, totalCredit);
        assertGt(val, 0, "Doesn't loop around");
    }

    function accountValue(bool isDebitNormal, uint128 totalDebit, uint128 totalCredit) public view returns (int128) {
        if (isDebitNormal) {
            // For debit-normal accounts: Value = Total Debit - Total Credit
            return int128(totalDebit) - int128(totalCredit);
        } else {
            // For credit-normal accounts: Value = Total Credit - Total Debit
            return int128(totalCredit) - int128(totalDebit);
        }
    }
}
```

Q-01 Int128 Removal, can delete 2 imports

https://github.com/centrifuge/protocol-v3/blob/e384a87a32f3a2e313a326ffe16c0a8f917620c9/src/misc/libraries/MathLib.sol#L178-L184

```
/// @notice Safe type conversion from uint256 to int128.
function toInt128(uint256 value) internal pure returns (int128) {
    require(value <= uint128(type(int128).max), Int128_Overflow());
    return int128(uint128(value));
}
```

Is no longer used

This comment can be removed

https://github.com/centrifuge/protocol-v3/blob/e384a87a32f3a2e313a326ffe16c0a8f917620c9/src/hub/Holdings.sol#L17-L18

```
using MathLib for uint256; // toInt128()
```

The function could also be deleted

Q-02 Small notes on Vault Router

QA: Some tokens cannot use this

https://github.com/centrifuge/protocol-v3/blob/e384a87a32f3a2e313a326ffe16c0a8f917620c9/src/vaults/VaultRouter.sol#L303-L309

```
    function _approveMax(address asset, uint256 tokenId, address spender) internal {
        if (tokenId == 0 && IERC20(asset).allowance(address(this), spender) == 0) {
            SafeTransferLib.safeApprove(asset, spender, type(uint256).max); /// @audit some tokens
cannot use this
        } else if (tokenId != 0 && IERC6909(asset).allowance(address(this), spender, tokenId) == 0)
{
            IERC6909(asset).approve(spender, tokenId, type(uint256).max);
        }
    }
```

- f000000000000 type scheme
- u88 and u128 (Uniswap and COMP)

QA - Balanche check is somewhat superfluous, should IMO simplify this

https://github.com/centrifuge/protocol-v3/blob/e384a87a32f3a2e313a326ffe16c0a8f917620c9/src/vaults/VaultRouter.sol#L135-L140

```
        if (vaultDetails.isWrapper && assetBalance < amount) {
            wrap(vaultDetails.asset, amount, address(this), msg.sender);
            lockDepositRequest(vault, amount, msg.sender, address(this));
        } else {
            lockDepositRequest(vault, amount, msg.sender, msg.sender);
        }
```

Add a flag for the conditional wrapping IMO `vaultDetails.isWrapper` && wrap -> Wrap -> Lock

Else just Lock

QA - Gotcha - Wouldn't fix

Seems like I can donate to a controller (owner = msg.sender, controller = other) But I wouldn't be able to fulfill their full path, as I wouldn't necessarily have `_canClaim`

I think this is fine as is

QA - Some functions are payable but don't use msg.value

Q-03 Insolvency / Account Breaking Operations

- Admin makes a mistake and fulfills more than `state.pendingRedeemRequest` in `fulfillCancelRedeemRequest`
- `triggerRedeemRequest` happens but no shares are moved into escrow
- The Admin can forget to increase assets in `approveDeposits` and in `revokeShares`, this will break all accounting
- The admin can `withdraw` and `deposit` in ways that move assets between `scId` and also in ways that steal the assets and break accounting

Q-04 `AsyncRequests.mint` and `AsyncRequests.withdraw`, `_calculateAssets` should `roundUp`

Impact

From first principles, when minting X shares we shoud `roundUp` the amount of assets necessary When withdrawing X assets, we shoud `roundUp` the amount of shares necessary

`withdraw` and `mint` in `AsyncRequests` don't follow these principles

https://github.com/centrifuge/protocol-v3/blob/25a9dc83a4953255364df96db5266ffaed3b0771/src/vaults/AsyncRequests.sol#L315-L316

```
        assets = uint256(_calculateAssets(vault_, shares_, state.depositPrice,
MathLib.Rounding.Down));
```

https://github.com/centrifuge/protocol-v3/blob/25a9dc83a4953255364df96db5266ffaed3b0771/src/vaults/AsyncRequests.sol#L365-L366

```
        shares = uint256(_calculateShares(vault_, assets_, state.redeemPrice,
MathLib.Rounding.Down));
```

It's worth noting that the entirety of the codebase is consistent with this decision (see `maxWithdraw` etc..), meaning this change should be investigated further

Mitigation

Review rounding to be consistent with the following:

- I will get a roundDown of shares when I pass assets (deposit)
- I will receive a roundDown of assets when I pass shares (redeem)
- I will pay a roundUp of assets when I mint shares (mint)
- I will pay a roundUp of shares when I withdraw assets (withdraw)

Q-05 Share token starting with no `hook` defaults to allowing everyone

Impact

https://github.com/centrifuge/protocol-v3/blob/c50ee4c37680d6b15951f998898dea8998c06973/src/vaults/token/ShareToken.sol#L21-L37

```
contract ShareToken is ERC20, IShareToken {
    using MathLib for uint256;

    mapping(address => Balance) private balances;

    /// @inheritdoc IShareToken
    address public hook;

    /// @inheritdoc IERC7575Share
    mapping(address asset => address) public vault;

    constructor(uint8 decimals_) ERC20(decimals_) {}

    modifier authOrHook() {
        require(wards[msg.sender] == 1 || msg.sender == hook, NotAuthorizedOrHook());
        _;
    }
```

https://github.com/centrifuge/protocol-v3/blob/c50ee4c37680d6b15951f998898dea8998c06973/src/vaults/token/ShareToken.sol#L139-L147

```

    function detectTransferRestriction(address from, address to, uint256 value) public view returns (uint8) {
        address hook_ = hook;
        if (hook_ == address(0)) return SUCCESS_CODE_ID;
        return IHook(hook_).checkERC20Transfer(from, to, value, HookData(hookDataOf(from), hookDataOf(to)))
            ? SUCCESS_CODE_ID
            : ERROR_CODE_ID;
    }
```

Means that it will return SUCCESS for all users at the beginning

Mitigation

Acknowledge or add a input in the constructor

Q-06 `_priceAssetPerShare` is subject to Oracle Drift Arbitrage when `valuation` uses an oracle and is combined with Synchronous Deposits

Impact

`valuation` allows the conversion between a `depositAsset` and a `poolAsset` which is then denominated in `poolAssetPerShare` which results in receiving a certain amount of shares with a corresponding `poolAsset` underlying amount

Whenever a conversion happens as the result of an oracle it will be subject to Oracle Drift

Definitions

Oracles are inherently inaccurate and for gas reasons (and precision / accuracy limitations) they can only update up to a certain precision

Meaning there is a deviation at which there is a difference between the Oracle Reported Price and the Market Price of an asset

I called this difference Oracle Drift

https://github.com/centrifuge/protocol-v3/blob/c50ee4c37680d6b15951f998898dea8998c06973/src/vaults/SyncRequests.sol#L352-L373

```
    function _priceAssetPerShare(
        PoolId poolId,
        ShareClassId scId,
        AssetId assetId,
        address asset,
        uint256 tokenId,
        IERC7726 valuation_
    ) internal view returns (D18 price) {
        if (address(valuation_) == address(0)) {
            (price,) = poolManager.priceAssetPerShare(poolId, scId, assetId, true);
        } else {
            IShareToken shareToken = poolManager.shareToken(poolId, scId);

            uint128 assetUnitAmount = uint128(10 ** VaultPricingLib.getAssetDecimals(asset, tokenId));
            uint128 shareUnitAmount = uint128(10 ** IERC20Metadata(shareToken).decimals());
            uint128 assetAmountPerShareUnit = /// @audit this is subject to oracle drift
                valuation_.getQuote(shareUnitAmount, address(shareToken), asset).toUint128();

            // Retrieve price by normalizing by asset denomination
            price = d18(assetAmountPerShareUnit, assetUnitAmount);
        }
    }
```

Given that you'd be allowing for instant deposits on vaults that are denominated in different assets this will be subject to arbitrage

POC - Depeg Insolvency Case

Assume I can convert from USDC to DAI to Pooled DAI

Assume USDC depegs to 85 cents

The oracle is not updated yet

I can now mint 1 Pooled DAI for 85 cents

POC - Arbitrage Case

Assume the oracle has a 2% deviation threshold

Assume USDC is trading at 98 cents of a DAI, I can buy it and mint 1 DAI worth of Pooled DAI, until the oracle updates

This opens up to a risk free arbitrage against other Vault Depositors, whom are socializing the value of their DAI

POC - Yield Frontrun case

This case applies even when `valuation` is only used to price `assets` being deposited against `assets * pps` in the pool

When a sufficiently high price appreciation is about to happen, then some depositors may elect to quickly mint and receive that additional appreciation

Having a long enough delay for withdrawals that ensures these depositors end up socializing some of their gains back is typically sufficient to prevent abuse, however this dynamic should be monitored

Mitigation

Syncronous Deposits must not convert from a currency to another, alternatively they should have a sufficiently high minting fee

Also keep in mind that being exposed to other currencies subjects the system to risking insolvency

Any currency<->currency conversion should be rate limited and should have a circuit breaker

Q-07 `BaseVauls.deposit` can be refactored to have CEI conformity

Impact

`BaseVauls.deposit` is transferring the token before calling `deposit`

https://github.com/centrifuge/protocol-v3/blob/7fbcf677dab86d94e07c1feea62d387750baa019/src/vaults/BaseVaults.sol#L350-L354

```
    function deposit(uint256 assets, address receiver) external returns (uint256 shares) {
        SafeTransferLib.safeTransferFrom(asset, msg.sender, syncDepositManager.escrow(), assets);
        shares = syncDepositManager.deposit(address(this), assets, receiver, msg.sender);
        emit Deposit(receiver, msg.sender, assets, shares);
    }
```

In order to comply with CEI it's best to swap these

Mitigation

```
    function deposit(uint256 assets, address receiver) external returns (uint256 shares) {
        shares = syncDepositManager.deposit(address(this), assets, receiver, msg.sender);
        SafeTransferLib.safeTransferFrom(asset, msg.sender, syncDepositManager.escrow(), assets);
        emit Deposit(receiver, msg.sender, assets, shares);
    }
```

Q-08 `price` rounding can cause very inaccurate results under extreme scenarios

Impact

`price` is calculated with this formula:

https://github.com/centrifuge/protocol-v3/blob/b3ebfede903e0dc32aa886ec82c4ae44ae514eaf/src/vaults/libraries/VaultPricingLib.sol#L60-L74

```
    function calculatePrice(address shareToken, uint128 shares, address asset, uint256 tokenId, uint128 assets)
        internal
        view
        returns (uint256)
    {
        if (assets == 0 || shares == 0) {
            return 0;
        }

        uint8 assetDecimals = getAssetDecimals(asset, tokenId);
        uint8 shareDecimals = IERC20Metadata(shareToken).decimals();
        return toPriceDecimals(assets, assetDecimals).mulDiv(
            10 ** PRICE_DECIMALS, toPriceDecimals(shares, shareDecimals), MathLib.Rounding.Down
        );
    }
```

This is always rounding down

In many scenarios, the truncation will lead to underpricing shares, by 1 wei over it's denomination

This causes the value to underprice on withdrawals (correct) but also underprice on deposits (socializes some yield)

In most scenarios the impact is marginal for 2 reasons:

- The loss would be 1/1e18 in order of magnitude
- Price per share is a hardcoded value that cannot be manipulated via external calls

In some scenario the impact can be quite significant

From my understand these extreme scenarios are highly unlikely when Assets and Shares are all read through storage values

Below I show a collection of extreme edge cases, worth reviewing and investigating further

It's worth noting that because of this discrepancy between the "real price per share" and the "cached price per share" It may be unsafe to use these vaults as collateral for lending protocols when the price per share value is manipulatable by user action

Methodology

See: https://github.com/Recon-Fuzz/centrifuge-hack

POC

```
    // forge test --match-test test_optimize_shares_upper_18_18_down_36 -vvv
    function test_optimize_shares_upper_18_18_down_36() public {
        // 108522494616216426649955010767567146266 || 108e36
        test_compare_calculateShares(217044989232432853734, 1, 500000000000000001, 17, 53, false);
        console2.log("optimize_shares_upper_18_18_down", optimize_shares_upper_18_18_down());
    }
```

```
Ran 1 test for test/recon/CryticToFoundry.sol:CryticToFoundry
[PASS] test_optimize_shares_upper_18_18_down_36() (gas: 104916)
Logs:
  totalVaultAssets 1
  assetDecimals 18
  totalVaultShares 500000000000000001
  shareDecimals 18
  price 1
  shares 217044989232432853734000000000000000000
  fullMathEvenMoreAccurateWithPrice 108522494616216427084044989232432853734
  optimize_shares_upper_18_18_down 108522494616216426649955010767567146266
```

Mitigation

This is a full precision formula

```
        uint256 assets = uint256(shares).mulDiv(totalVaultAssets * 1, totalVaultShares * 1,
rounding);
```

```
        uint256 shares = uint256(
            uint256(assets) * uint256(totalVaultShares)
            ).mulDiv(1, uint256(totalVaultAssets) * 1, rounding);
```

The risk with this formula is a lack of normalization on the first mint

I believe once assets have been deposited and shares have been minted, that this formula will work correctly

I'm not as confident when there are zero deposits, for that case, you may either chose to use 10**Decimals or you may want to normalize everything to 18 decimals

Please keep in mind that if you will retain `price` then you won't be able to add the additional precision as you need the two components being separate in order to not lose any precision due to truncation

Q-09 Multiple theoretical issues with the `_updatePending` vs `_updateQueued` logic

Impact

In order to simplify accounting, whenever an `investor` has already performed a deposit request before having that processed, their next request will be automatically queued

My understanding is that this makes it so that the call to `approveDeposits` is going to have less race conditions

It's worth noting that because `approveDeposits` uses `maxApproval`, the only possible impact of the race condition is that we won't be able to determine which `investor` get's what % of their request approved

As some other investor may quickly front-run the admin to get some of their request approved

Queueing logic:

```
        if (userOrder.lastUpdate > latestApproval || userOrder.pending == 0 || latestApproval == 0)
{
            return false;
        }
```

Given this, the following "attacks" and "gotchas" are possible:

- Edge case on first epoch | On first epoch all investors will be able to spam requests and they will not be queued

- There is no guarantee that when `userOrder.lastUpdate > latestApproval` the value is empty, due to the `(claimableShareAmount > 0) {` edge case in `claimDepositUntilEpoch` and because claiming deposits calls `_postClaimUpdateQueued` which updates the `userOrder.pending` without updating the `userOrder.lastUpdate` to a future epoch

- When an `investor` has `userOrder.pending == 0` they can add requests directly, meaning a savy investor could use more than one account to skip getting queued

All of these observations lead me to believe that the queue is fundamentally not offering any specific advantage, while it's adding extra code

I would at this point recommend either fully enforcing the queue or getting rid of it

State Transitions

Given an admin that is about to `approveX`

A investor can have it's `pending` update by an indefinite amount if they have had some amount added off of the queue, or if their `pending` is left from previous `approve`s

The amounts are also undefined for new investors since they can contribute any amount to the total `pendingX`

Q-10 `ShareClassManager.if (claimableAssetAmount > 0) {` is an implicit "limit order"

Summary

These are my thoughts around the lines:

https://github.com/centrifuge/protocol-v3/blob/b3ebfede903e0dc32aa886ec82c4ae44ae514eaf/src/hub/ShareClassManager.sol#L439-L462

```
            /// @audit an implicit stinky bid / limit order on saying that I can get something cheaper than current
            uint128 claimableAssetAmount = uint256(approvedShareAmount).mulDiv(
                epochAmounts_.redeemAssets, epochAmounts_.redeemApproved
            ).toUint128();

            // NOTE: During approvals, we reduce pendingRedeems by the approved share class token amount. However, we
            // only reduce the pending user amount if the claimable amount is non-zero.
            //
            // This extreme edge case has two implications:
            //  1. The sum of pending user orders <= pendingRedeems (instead of equality)
            //  2. The sum of claimable user amounts <= amount of payout asset corresponding to the approved share class
            // token amount (instead of equality).
            //     I.e., it is possible for an epoch to have an excess of a single payout asset unit which cannot be
            // claimed by anyone. This excess is at most n-1 payout asset units for an epoch with n claimable users.
            //
            // The first implication can be switched to equality if we reduce the pending user amount independent of the
            // claimable amount.
            // However, in practice, it should be extremely unlikely to have users with non-zero pending but zero
            // claimable for an epoch.
            if (claimableAssetAmount > 0) {
                paymentShareAmount += approvedShareAmount;
                payoutAssetAmount += claimableAssetAmount;
                userOrder.pending -= approvedShareAmount;
            }
```

Which effectively make some claims non deterministic

Most specifically, any time a `userOrder.pending` is not subtracted due to the edge case, the userOrder.pending can be requeued in a future epoch

This is equivalent to saying that the user is placing a limit order on getting some asset for less `pending`, since any time they would get an insufficient amount of asset they will have their request re-queued

It's worth noting that the comments around the code are correct: And I was unable to cause any insolvency through the logic As the logic is similar to queueing

Impact

This is a form of implicit queueing

As it fundamentally makes it so that a user.pending will be > 0 on the next epoch

I believe there are more implications from this, but they are tied to how the queueing system works in the SCM more so than this specific issue

Considerations

I believe the code will be fine as is, it's worth showing the edge case to future reviewers

We will also provide a simple suite that can reach coverage in less than 1 hour as a means to help auditors perform extensive DD on the code

Q-11 BaseInvestmentManager `convertToAssets` and `convertToShares` have 3 separate instances of rounding

Impact

The Code for `convertToShares` and `convertToAssets` is as follows:

https://github.com/centrifuge/protocol-v3/blob/1b0ee17f50c54bf8eba108736fa23a510e8bad74/src/vaults/BaseInvestmentManager.sol#L41-L59

```
    function convertToShares(address vaultAddr, uint256 assets) public view virtual returns
(uint256 shares) {
        IBaseVault vault_ = IBaseVault(vaultAddr);
        VaultDetails memory vaultDetails = poolManager.vaultDetails(address(vault_));
        (D18 priceAssetPerShare,) =
            poolManager.priceAssetPerShare(mapPoolId(vault_.poolId()), vault_.trancheId(),
vaultDetails.assetId, false);

        return _convertToShares(vault_, vaultDetails, priceAssetPerShare, assets,
MathLib.Rounding.Down);
    }

    /// @inheritdoc IBaseInvestmentManager
    function convertToAssets(address vaultAddr, uint256 shares) public view virtual returns
(uint256 assets) {
        IBaseVault vault_ = IBaseVault(vaultAddr);
        VaultDetails memory vaultDetails = poolManager.vaultDetails(address(vault_));
        (D18 priceAssetPerShare,) = // TODO: Rounding TWICE!!
            poolManager.priceAssetPerShare(mapPoolId(vault_.poolId()), vault_.trancheId(),
vaultDetails.assetId, false);

        return _convertToAssets(vault_, vaultDetails, priceAssetPerShare, shares,
MathLib.Rounding.Down);
    }
```

This is computing the `priceAssetPerShare` which consists of a truncation:

https://github.com/centrifuge/protocol-v3/blob/1b0ee17f50c54bf8eba108736fa23a510e8bad74/src/vaults/PoolManager.sol#L463-L464

```
        price = poolPerShare.asPrice() / poolPerAsset.asPrice(); /// @audit Rounds down
```

It then uses `VaultPricingLib` for which we already know rounding is applied

This means that currently the code is performing 3 rounding operations on any one convertion

If we fix #26 we will still have 2 rounding errors to deal with

Due to this I believe that the code should be changed to limit itself to one rounding error

This can be done by performing operations with higher precision (using `fullMath`), which should allow simplifying the various formulas down to simplified ones that also carry less precision issues

Q-12 `PoolManager.priceAssetPerShare` is always rounding the `priceAssetPerShare` down, which is not always correct

https://github.com/centrifuge/protocol-v3/blob/1b0ee17f50c54bf8eba108736fa23a510e8bad74/src/vaults/PoolManager.sol#L450-L465

```
    function priceAssetPerShare(uint64 poolId, bytes16 scId, uint128 assetId, bool checkValidity)
        public
        view
        returns (D18 price, uint64 computedAt)
    {
        (Price memory poolPerAsset, Price memory poolPerShare) = _poolPer(poolId, scId, assetId);

        if (checkValidity) {
            require(poolPerAsset.isValid(), InvalidPrice());
            require(poolPerShare.isValid(), InvalidPrice());
        }

        // (POOL_UNIT/SHARE_UNIT) / (POOL_UNIT/ASSET_UNIT) = ASSET_UNIT/SHARE_UNIT
        price = poolPerShare.asPrice() / poolPerAsset.asPrice(); /// @audit Rounds down
        computedAt = poolPerShare.computedAt;
    }
```

This formula will end up rounding down by 1 wei of a share per asset

When we add this into the context that we will use this value as a multiplicative value, then the error can be magnified

This will cause either a loss to users of that amount per each unit/share Or it will incorrectly discount each share by that amount per each share/unit

Mitigation

TODO: Many parts of the code should have one function with the highest precision and one rounding operation Having multiple rounding errors create VERY complex behaviour that opens up to a lot of tail risk

Q-13 `SyncRequests.convertToAssets` rounding direction is safe for `previewMint` but it's incorrect for the ERC4626 Spec

Impact

`previewMint` answer how many `assets` one must provide in order to receive `shares`

Rounding the `assets` `up` is the safest decision

However `convertToAssets` is another `public` function in the EIP4626 spec which states that `convertToAssets`:

> MUST round down towards 0

As such, the implementation is safe, and `previewMint` should keep this logic

https://github.com/centrifuge/protocol-v3/blob/1b0ee17f50c54bf8eba108736fa23a510e8bad74/src/vaults/SyncRequests.sol#L256-L270

```
    function convertToAssets(address vaultAddr, uint256 shares) /// @audit wrong but also used right, bit of a pickle
        public
        view
        override(IBaseInvestmentManager, BaseInvestmentManager)
        returns (uint256 assets)
    {
        IBaseVault vault_ = IBaseVault(vaultAddr);
        VaultDetails memory vaultDetails = poolManager.vaultDetails(address(vault_));
        D18 priceAssetPerShare_ = _priceAssetPerShare(
            vault_.poolId(), vault_.trancheId(), vaultDetails.assetId, vault_.asset(), vaultDetails.tokenId
        );
        /// @audit Wrong rounding
        return super._convertToAssets(vault_, vaultDetails, priceAssetPerShare_, shares, MathLib.Rounding.Up);
    }
```

However, you should either acknowledge and document this discrepancy or refactor so that the public `convertToAssets` rounds down

Q-14 ABA: Rounding direction is lost when `fromPriceDecimals` causes truncation

Impact

The finding was reported for Centrifuge V2 by ABA: https://github.com/abarbatei/audits

The following Foundry test demonstrates the issue:

```
function test_shares_precision(
    ) public {

        uint8 shareDecimals = 10;
        uint8 assetDecimals = 18;
        uint128 price = 1e18 - 1;
        uint128 assets = 1e18 - 3;


        uint256 sharesInPriceDecimals =
            toPriceDecimals(assets, assetDecimals).mulDiv(10 ** PRICE_DECIMALS, price, MathLib.Rounding.Down);
        console2.log("sharesInPriceDecimals", sharesInPriceDecimals);

        console2.log("sharesInPriceDecimals 18 ", fromPriceDecimals(sharesInPriceDecimals, 18));
        console2.log("sharesInPriceDecimals 10 ", fromPriceDecimals(sharesInPriceDecimals, 10)); // THIS IS ALWAYS ROUNDING DOWN DUE TO TRUNCATION

        // THIS IS ROUNDING CORRECTLY
        // TODO: The rounding MUST be applied on the last operation
        // TODO: Consider using full precision (mulDiv is already doing that) and perform operations in bulk rather than separately
        console2.log("Full Precision? ", toPriceDecimals(assets, assetDecimals).mulDiv(10**10, price, MathLib.Rounding.Up));
    }
```

With logs

```
[PASS] test_shares_precision() (gas: 9482)
Logs:
  sharesInPriceDecimals 999999999999999997
  sharesInPriceDecimals 18  999999999999999997
  sharesInPriceDecimals 10  9999999999
  Full Precision?  10000000000
```

Whenever the call to `fromPriceDecimals` causes a truncation (which happens whenever shares use decimals lower than 18), the call to `fromPriceDecimals` will truncate some digits

Since the rounding is computed on the previous calculation, then the truncation will eliminate the rounding, causing all operations to round down at all times

Escalating the Impact

We need 2 preconditions:

- Shares have less than 18 decimals
- We need an operation that is roundingUp

TODO: THEORETIC

I'd expect this to be the case for `mint` since mint allows us to specify a certain amount of shares and should then `roundUp` the assets we need to provide

This should give us a 1 wei share discount, which over time can be significant

This can become even more significant if we can find ways to rebase the shares

TODO: NEED TO CHECK

Q-15 `Hub.updateHoldingAmount` introduces a lot of complexity but can only be used in scenarios that can be executed with simpler functions

Impact

`updateHoldingAmount` looks as follows:

https://github.com/centrifuge/protocol-v3/blob/b435fa2858ce7ac2f519f2d17896040f911eacc5/src/hub/Hub.sol#L445-L473

```
/// @inheritdoc IHubGatewayHandler
    function updateHoldingAmount(
        PoolId poolId,
        ShareClassId scId,
        AssetId assetId,
        uint128 amount,
        D18 pricePoolPerAsset,
        bool isIncrease,
        JournalEntry[] memory debits,
        JournalEntry[] memory credits
    ) external {
        _auth();

        accounting.unlock(poolId);

        address poolCurrency = hubRegistry.currency(poolId).addr();
        transientValuation.setPrice(assetId.addr(), poolCurrency, pricePoolPerAsset);
        uint128 valueChange = transientValuation.getQuote(amount, assetId.addr(),
poolCurrency).toUint128();

        (uint128 debited, uint128 credited) = _updateJournal(debits, credits);
        uint128 debitValueLeft = valueChange - debited;
        uint128 creditValueLeft = valueChange - credited;

        _updateHoldingWithPartialDebitsAndCredits(
            poolId, scId, assetId, amount, isIncrease, debitValueLeft, creditValueLeft
        );

        accounting.lock();
    }
```

The function is pretty complex, with:

- Adding amounts
- Applying journals
- Using those values to apply `debitValueLeft` and `creditValueLeft`

However, as of now updates to the `accounting` must be balanced

As such this function can only be used when `debitValueLeft == creditValueLeft`

Making the operation identical to simpler:

- Deposit / Withdraw
- Update accounts

Operations

POC

To emulate the behaviour I wrote this simplified version

This will digest all `JournalEntry` to one addition of `debits` and one of `credits`)

I'm following the logic from the source and then asserting that the operation, that looks pretty complex, is in fact very simple:

```
eq(debitValueLeft, creditValueLeft, "We can never go past this with different values");
```

```
function updateHoldingAmount(
        uint128 amount,
        // D18 pricePerUnit, NOTE: We skip updating price, it's updated by TargetFunctions
        bool isIncrease,
        uint256 debitIndex,
        uint256 creditIndex,
        uint128 debits,
        uint128 credits
    ) public {
        _unlock();

        // NOTE: Addresses are ignored
        uint256 fullPrecisionChange = transientValuation.getQuote(amount, address(0), address(0));
        require(fullPrecisionChange <= type(uint128).max, "Full precision change is too large");
        uint128 valueChange = uint128(fullPrecisionChange);

        /// @audit the update here desynchs the holdings as the holdings remain unchanged, but we add values

        // Apply some debit and some credits
        // Could be imbalanced | TODO: Which accounts can we use?
        (uint128 debited, uint128 credited) = _updateJournal(debitIndex, debits, creditIndex, credits);
        uint128 debitValueLeft = valueChange - debited;
        uint128 creditValueLeft = valueChange - credited;

        _updateHoldingWithPartialDebitsAndCredits(
            amount,
            isIncrease,
            debitValueLeft,
            creditValueLeft
        );

        _lock();
        // TODO: Remove stateless and allow proper tracking

        eq(debitValueLeft, creditValueLeft, "We can never go past this with different values");

        revert("stateless");
    }
    function _updateHoldingWithPartialDebitsAndCredits(
        uint128 amount,
        bool isIncrease,
        uint128 debitValue,
        uint128 creditValue
    ) internal {
        bool isLiability = holdings.isLiability(poolId, scId, payoutAssetId); // False all the time
        t(!isLiability, "isLiability"); // Always false

        //
        // AccountType debitAccountType = isLiability ? AccountType.Expense : AccountType.Asset;
        // AccountType creditAccountType = isLiability ? AccountType.Liability : AccountType.Equity;


        // Add X, Remove Y
        /// Add Z - X on one side, add Z + Y  on the other, I think either there's zeros or they must be the same value

        if (isIncrease) {
            holdings.increase(poolId, scId, payoutAssetId, transientValuation, amount); /// @audit I think this breaks the property
            // accounting.addDebit(, debitValue);
            // accounting.addCredit(, creditValue);

            // TODO: Are these correct? NOTE: NO THIS CAN'T BE
            // TODO: How would this become correct? TOOD: Go back to properties
            // depositValue = depositValue + (int256(uint256(creditValue)) - int256(uint256(debitValue)));
            yieldValue = yieldValue + (int256(uint256(creditValue)) - int256(uint256(debitValue)));

            // Increase in equity, decrease in asset?
            accounting.addCredit(EQUITY_ACCOUNT, creditValue);
            accounting.addDebit(ASSET_ACCOUNT, debitValue); /// @audit this could be both higher or lower

        } else {
            // Loss of equity, increase in asset?
            holdings.decrease(poolId, scId, payoutAssetId, transientValuation, amount);
            accounting.addCredit(ASSET_ACCOUNT, creditValue);
            accounting.addDebit(EQUITY_ACCOUNT, debitValue);

            // depositValue = depositValue + ((debitValue)) - int256(uint256(creditValue)));
            yieldValue = yieldValue + (int256(uint256(debitValue)) - int256(uint256(creditValue)));
        }
    }
```

Mitigation

Consider either fully developing the function for a specific state transition

Or remove it

As it stands the function is very complex, but can only be used for balanced operations, meaning it can be rewritten to simply use the Journals and perform a deposit or a withdrawal

Q-16 Queueing can be sidestepped with multiple accounts

Impact

Queueing can only happen for accounts that have a non zero `userOrder.pending` on an epoch past 1

By creating a new account and depositing or by using a second account, we can sidestep this

Q-17 `userOrder.pending` update in `_updatePending` can be simplified

Impact

`userOrder.pending` is updated in `_updatePending` in this way

```
        userOrder.pending = isIncrement ? userOrder.pending + amount : userOrder.pending - amount;
```

However, we know that when we are performing a decrement the value will decrease by 100%

As such we could simplify the code here following that idea

```
userOrder.pending = isIncrement ? userOrder.pending + amount : 0;
```

Q-18 Multiple `requestDeposit` and one `cancelDeposit` can result in undefined behaviour

Impact

I believe this is mostly a known issue, just adding it for future reference

I also recommend other reviewers to see if my logic makes sense

As a user I can:

- Queue more than one deposit
- Request a cancellation over one or more deposits

Due to a race condition, once a cancellation has been queued, it will force the requesting user to have that cancelled

Meaning that if:

- The requests are fulfilled out of order
- The messaging layer allows replays (it should)

The user request can result in execution that is out of order

I'm not fully sure if this is wholly solvable

It's worth noting that as long as every user has at most:

- 1 Deposit Request to be Fulfilled
- 1 Cancellation Request to be Fulfilled

Active at any time, then executions ordering will be fairly consistent

```
// SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

import {FoundryAsserts} from "@chimera/FoundryAsserts.sol";

import {Test, console2} from "forge-std/Test.sol";

import {TargetFunctions} from "./TargetFunctions.sol";
import {Helpers} from "test/pools/fuzzing/recon-pools/utils/Helpers.sol";

import {AssetId} from "src/common/types/AssetId.sol";


// forge test --match-contract CryticToFoundry --match-path test/pools/fuzzing/recon-scm-
only/CryticToFoundry.sol -vv
contract CryticToFoundry is Test, TargetFunctions, FoundryAsserts {

    function setUp() public {
        setup();
    }



    function test_basic() public {
      shareClassManager_requestDeposit(123);
      _switchActor(1);
      shareClassManager_requestDeposit(123);
      shareClassManager_approveDeposits(123);
      _logRequests();

      _switchActor(0);
      shareClassManager_cancelDepositRequest();
      _logRequests();
      // They cannot queue again since they requested cancelling
    //   shareClassManager_requestDeposit(123);
    //   _logRequests();
    }

    uint256 count;
    function _logRequests() public {
        console2.log("");
        console2.log("");
        console2.log("_logRequests", count++);

        {
            (uint128 depositRequest, uint32 lastUpdate) = shareClassManager.depositRequest(scId,
depositAssetId, bytes32(uint256(uint160(_getActor()))));
            console2.log("depositRequest", depositRequest);
        }

        {
            (uint128 redeemRequest, uint32 lastUpdateRedeem) = shareClassManager.redeemRequest(scId,
depositAssetId, bytes32(uint256(uint160(_getActor()))));
            console2.log("redeemRequest", redeemRequest);
        }

        {
            (bool isCancelling, uint128 queuedDepositRequest) =
shareClassManager.queuedDepositRequest(scId, depositAssetId,
bytes32(uint256(uint160(_getActor()))));
            console2.log("queuedDepositRequest", queuedDepositRequest);
        }
        {
            (bool isCancelling, uint128 queuedRedeemRequest) =
shareClassManager.queuedRedeemRequest(scId, depositAssetId,
bytes32(uint256(uint160(_getActor()))));
            console2.log("queuedRedeemRequest", queuedRedeemRequest);
        }
    }
}
```

Q-19 QA Findings - SCM

Simplify Require Statement

```
        require(!(queued.isCancelling == true && amount > 0), CancellationQueued());
        // require(queued.isCancelling == false || amount == 0, CancellationQueued()); // Simpler to reason around
```

Q-20 `updateHoldingValue / updateJournal` can cause socialized losses if the admin doesn't `updateHoldingValuation` when shares value rapidly changes

Impact

The double entry bookeping of the system seems safe, in the sense that it's always balanced barring admin mistake (or maliciousnes)

However, while the absolute values are not subject to race conditions (as they ultimately lead to the same result), the multiplicative ratio of these value does

For example, the underlying assets getting a haircut of 25% can result in a 25% decrease in the value in the system or a 31.25% decrease (an example) if more withdrawals were processed before accounting for the haircut

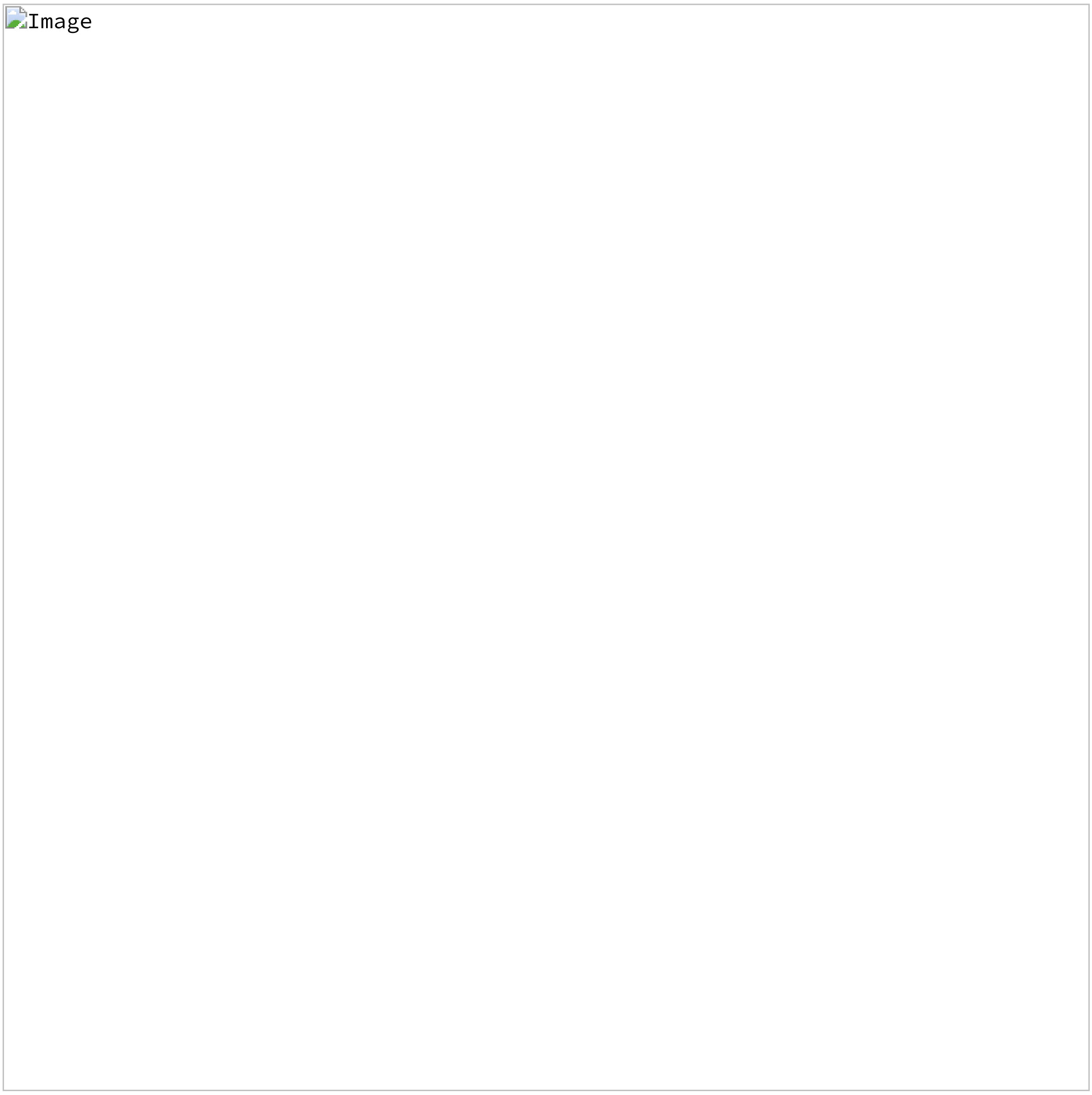
I have yet to fully understand if this can have impacts on share math as that would be the variable that most likely could be impacted

Logical Code Path

A haircut on valuation is known User request redeems User receive revoked shares payout Admin doesn't socialize the loss (uses cached `navPerShare`) User exits without the loss updateHoldingValuation is called, socializing the loss, including the loss from the user that exited

So this requires the admin to make a mistake in the order of operations, and to not update the valuation for all shares before processing the withdrawal

Example



Example Case

Loans underlying a set of bonds will default

Causing a haircut to the value of the shares

The haircut is a percentage for all shares

Some users `requestRedeem` as a means to not be part of that loss

Their request is fulfilled

The loss to the underlying Assets is computed in absolute terms, meaning that the loss will result in a higher % loss to those that didn't `redeem`

Loss was socialized to those that maintained their position as the other holders were able to redeem a value that didn't include the losses

Mitigation

This issue falls into race conditions that can be known externally but cannot by known by the vault

Ultimately every Vault Manager should define a specific policy on how they would behave in this scenario (e.g. all withdrawals will be blocked until the new valuation is on-hain)

The steps taken by the Vault Manager would completely eliminate any risk of loss socialization, or at the very least they would make this scenario predictable as the Manager would act based on a publicly known process

Q-21 Gas Optimizations - SCM

Change order of operations for `_updatePendingDeposit` and `_updatePendingRedeem`

Improves legibility and saves 100 gas

```
    function _updatePendingRedeem(
        PoolId poolId,
        ShareClassId scId_,
        uint128 amount,
        bool isIncrement,
        bytes32 investor,
        AssetId assetId,
        UserOrder storage userOrder,
        QueuedOrder memory queued
    ) private {
        uint128 pendingTotal = pendingRedeem[scId_][assetId]; /// @audit Gas and Simplicity? | // 1 read
        pendingRedeem[scId_][assetId] = isIncrement ? pendingTotal + amount : pendingTotal - amount; // 1 write
        pendingTotal = pendingRedeem[scId_][assetId]; // 2nd Read
    }

// 2 Read 1 Write
```

Change to

```
        uint128 pendingTotal = pendingDeposit[scId_][assetId]; /// @audit Save to memory
        pendingTotal = isIncrement ? pendingTotal + amount : pendingTotal - amount; /// @audit update memory
        pendingDeposit[scId_][assetId] = pendingTotal; /// Update Storage

// 1 Read, 1 write
```

You don't need to compare a `bool` with `true`

https://github.com/centrifuge/protocol-v3/blob/95b10b57af336a111ac9f82c25caf5dc0b0310e6/src/hub/ShareClassManager.sol#L751-L752

```
require(!(queued.isCancelling == true && amount > 0), CancellationQueued());
```

Can be changed to

```
require(!(queued.isCancelling && amount > 0), CancellationQueued());
```

Q-22 Permit wont' work with DAI

https://github.com/centrifuge/protocol-v3/blob/95b10b57af336a111ac9f82c25caf5dc0b0310e6/src/vaults/VaultRouter.sol#L278-L284

```
    function permit(address asset, address spender, uint256 assets, uint256 deadline, uint8 v, bytes32 r, bytes32 s)
        external
        payable
        protected
    {
        try IERC20Permit(asset).permit(msg.sender, spender, assets, deadline, v, r, s) {} catch {}
    }
```

IMO safe to Ack

Q-23 It's generally best not to grant permissioneless execution

Impact

In many cases permissioneless execution can open up to MEV attack

It's generally best to not grant permissioneless execution, unless you can explain explicitly why the execution is safe

https://github.com/centrifuge/protocol-v3/blob/95b10b57af336a111ac9f82c25caf5dc0b0310e6/src/common/Root.sol#L95-L104

```
    function executeScheduledRely(address target) external {
        require(schedule[target] != 0, TargetNotScheduled());
        require(schedule[target] <= block.timestamp, TargetNotReady());

        wards[target] = 1;
        emit Rely(target);

        schedule[target] = 0;
    }
```

Mitigation

Either document why this is safe or consider adding back `auth`

Q-24 `receiveWormholeMessages` is `payable` but the value is unused

https://github.com/centrifuge/protocol-v3/blob/95b10b57af336a111ac9f82c25caf5dc0b0310e6/src/common/WormholeAdapter.sol#L46-L59

```
function receiveWormholeMessages(
    bytes memory payload,
    bytes[] memory, /* additionalVaas */
    bytes32 sourceAddress,
    uint16 sourceWormholeId,
    bytes32 /* deliveryHash */
) external payable {
    WormholeSource memory source = sources[sourceWormholeId];
    require(source.addr == sourceAddress.toAddressLeftPadded(), InvalidSource());
    require(msg.sender == address(relayer), NotWormholeRelayer());

    gateway.handle(source.centrifugeId, payload);
}
```

Q-25 `Permissioneless Methods` lists `createPool` which has the modifier `auth`

https://github.com/centrifuge/protocol-v3/blob/95b10b57af336a111ac9f82c25caf5dc0b0310e6/src/hub/Hub.sol#L112-L124

```
    //-----------------------------------------------------------------------------------------------
    // Permisionless methods
    //-----------------------------------------------------------------------------------------------

    /// @inheritdoc IHub
    function createPool(address admin, AssetId currency)
        external
        payable
        auth
        returns (PoolId poolId)
    {
        poolId = hubRegistry.registerPool(admin, sender.localCentrifugeId(), currency);
    }
```

Q-26 `_generateJournalId` can be changed to make full use of it's type

Impact

https://github.com/centrifuge/protocol-v3/blob/95b10b57af336a111ac9f82c25caf5dc0b0310e6/src/hub/Accounting.sol#L110-L113

```
    /// @audit QA: Shouldn't this be uint256(poolId.raw()) << 128? So it's shifted up by 128 and uses the full type?
    function _generateJournalId(PoolId poolId) internal returns (uint256) {
        return uint256((uint128(poolId.raw()) << 64) | ++_poolJournalIdCounter[poolId]);
    }
```

Refactor

```
    function _generateJournalId(PoolId poolId) internal returns (uint256) {
        return uint256((uint256(poolId.raw()) << 128) | ++_poolJournalIdCounter[poolId]);
    }
```

Will make full use of the type Do test it with Halmos for all poolJournalValues below uint128!

Q-27 `Holdings.update` can revert when the absolute difference in value is above `type(int128).max`

Impact

The `diffValue` can enter a reverting state whenever the `abs(currentAmountValue) - abs(assetAmountValue` is higher than `type(int128).max`

This is very low likelihood scenario

But it will cause a revert

I believe that chunking the delta of changes into 2 updates would unstuck the system

https://github.com/centrifuge/protocol-v3/blob/4da9ab04e637bcc31ae305347e42826dcbb8908d/src/pools/Holdings.sol#L95-L111

```
    function update(PoolId poolId, ShareClassId scId, AssetId assetId) external auth returns
(int128 diffValue) {
        Holding storage holding_ = holding[poolId][scId][assetId];
        require(address(holding_.valuation) != address(0), HoldingNotFound());
        /// @audit This could cause revert DOS, but not a overflow
        uint128 currentAmountValue = holding_.valuation.getQuote(
            holding_.assetAmount, assetId.addr(), poolRegistry.currency(poolId).addr()
        ).toUint128();

        diffValue = currentAmountValue > holding_.assetAmountValue
            ? uint256(currentAmountValue - holding_.assetAmountValue).toInt128()
            : -uint256(holding_.assetAmountValue - currentAmountValue).toInt128();

        holding_.assetAmountValue = currentAmountValue;

        emit Update(poolId, scId, assetId, diffValue);
    }
```

POC / IMPACT

```
    /**
        function update(PoolId poolId, ShareClassId scId, AssetId assetId) external auth returns
(int128 diffValue) {
            Holding storage holding_ = holding[poolId][scId][assetId];
            require(address(holding_.valuation) != address(0), HoldingNotFound());
            uint128 currentAmountValue = holding_.valuation.getQuote(
                holding_.assetAmount, assetId.addr(), poolRegistry.currency(poolId).addr()
            ).toUint128();

            diffValue = currentAmountValue > holding_.assetAmountValue
                ? uint256(currentAmountValue - holding_.assetAmountValue).toInt128()
                : -uint256(holding_.assetAmountValue - currentAmountValue).toInt128();

            holding_.assetAmountValue = currentAmountValue;

            emit Update(poolId, scId, assetId, diffValue);
        }
     */

    error Int128_Overflow();
    function toInt128(uint256 value) internal pure returns (int128) {
        if(value > uint128(type(int128).max)) {
            assert(false);
        }
        return int128(uint128(value));
    }
    function test_update_math(uint128 currentValue, uint128 previousValue) public {
    /**
        Revert Case 1:
        Delta is above int128.max
        [FAIL: Int128_Overflow(); counterexample:
calldata=0xc759c44800000000000000000000000000000000fffffffffffffffffffffffffffffffd00000000000000000
00000000000000000000000000000923c49204e6b3bd517 args=[340282366920938463463374607431768211453
[3.402e38], 2697568673598348711191 [2.697e21]]] test_update_math(uint128,uint128) (runs: 3, μ: 999,
~: 952)
     */
        vm.assume(currentValue < uint128(type(int128).max));
        vm.assume(previousValue < uint128(type(int128).max));

        int128 diffValue = currentValue > previousValue
            ? toInt128(uint256(currentValue - previousValue))
            : -toInt128(uint256(previousValue - currentValue));


    }
```

```
[PASS] test_update_math(uint128,uint128) (paths: 6, time: 0.05s, bounds: [])
Symbolic test result: 1 passed; 0 failed; time: 0.19s
```

Whereas removing the `assume`s will result in reverts for the edge case discussed above

```
Running 1 tests for test/recon/CryticToFoundry.sol:CryticToFoundry
Counterexample:
    p_currentValue_uint128_8a1cc3c_00 = 0x80000000000000000000000000000001
    p_previousValue_uint128_029524b_00 = 0xc0b821100ffffe0000000000057fffff
    p_previousValue_uint128_029524b_00 = 0x00Counterexample:
    p_currentValue_uint128_8a1cc3c_00 = 0x1b8b113965820000000000008000001
```

Mitigation

Consider using `uint128` for all values

And if necessary use a `bool` to signify a negative value

Q-28 `previewMint` rounding direction is incorrect

Impact

`previewMint` is asking: How many `assets` do you have to provide to mint those `shares`

Due to this the code should round the `assets` up

https://github.com/centrifuge/protocol-v3/blob/fc98c473f222419d124fc6a53d53c2fee25ef2e4/src/vaults/SyncRequests.sol#L147-L159

```
    function previewMint(address vaultAddr, address, /* sender */ uint256 shares)
        public
        view
        returns (uint256 assets)
    {
        SyncDepositVault vault_ = SyncDepositVault(vaultAddr);
        uint128 assetId = poolManager.vaultDetails(vaultAddr).assetId;

        uint128 latestPrice = _pricePerShare(vaultAddr, vault_.poolId(), vault_.trancheId(), assetId);
        assets = PriceConversionLib.calculateAssets(shares.toUint128(), vaultAddr, latestPrice, MathLib.Rounding.Down);
    }
```

Shouldn't you pay MORE assets for each share?

Mitigation

Use `MathLib.Rounding.Up`

Q-29 `Vaults` `try/catch` may be subject to Low Gas Attack

Impact

A `try/catch` can substantially fail for 2 reasons:

- The Call Reverts because it was meant to
- The Call Reverts due to OOG

The code for `requestRedeem` looks as follows:

https://github.com/centrifuge/protocol-v3/blob/95b10b57af336a111ac9f82c25caf5dc0b0310e6/src/vaults/BaseVaults.sol#L220-L242

```
    function requestRedeem(uint256 shares, address controller, address owner) public returns (uint256) {
        require(IShareToken(share).balanceOf(owner) >= shares, "AsyncVault/insufficient-balance");

        // If msg.sender is operator of owner, the transfer is executed as if
        // the sender is the owner, to bypass the allowance check
        address sender = isOperator[owner][msg.sender] ? owner : msg.sender;

        require(
            asyncRedeemManager.requestRedeem(address(this), shares, controller, owner, sender),
            "AsyncVault/request-redeem-failed"
        );

        address escrow = asyncRedeemManager.escrow();
        try IShareToken(share).authTransferFrom(sender, owner, escrow, shares) returns (bool) {}
        catch {
            // Support share class tokens that block authTransferFrom. In this case ERC20 approval needs to be set
            require(IShareToken(share).transferFrom(owner, escrow, shares), "AsyncVault/transfer-from-failed");
        }

        emit RedeemRequest(controller, owner, REQUEST_ID, msg.sender, shares);
        return REQUEST_ID;
    }
```

Because of how gas metering works in the EVM, it may be possible to:

- Have the `authTransferFrom` be more expensive than the `transferFrom`
- Make the `transferFrom` substantially cheaper than the `authTransferFrom` due to having read the storage slots in the `authTransferFrom`

And while hard to fully prove without spending a substantial amount of time, in some calls, it may be possible to abuse the 1/64 rules to trigger the `transferFrom` clause when that's not intended

Mitigation

As far as I'm aware it is impossible to tell if a call has reverted because it was provided insufficient gas, without performing a check before the call

My recommendation is to enforce that `authTransferFrom` is always provided sufficient gas

See the work we did with Liquity: https://github.com/liquity/V2-gov/blob/d910db20c035c4d2c9b301a2f92cff1cde50f28c/src/Governance.sol#L125-L127

Alternative, stricter mitigation

If you wish to conditionally use `transferFrom` exclusively when the Hook causes a `RestrictionsFailed` error, then you should capture the revert reason

```
    try share.authTransferFrom(msg.sender, msg.sender, address(this), amount) returns (bool) {} // Capture this: RestrictionsFailed(), ensure it's the correct error and then perform the alternative swap
```

NOTE: Errors with no params are only 4 bytes so this is pretty cheap to do

A-01 Accounting Gotchas wrt processing yield, pending deposits and withdrawals

Impact

Because of the fact that there can be simultaneously multiple requests at a time

There's the following gotcha: -You could have assets that are yet to be added to any share, and that would possibly cause a incorrect positive rebase (see Sync Deposit)

- You could have assets that are removed as shares are being issued (which can lead to incorrect yield math as well as incorrect math in general)

This fundamentally means the following:

- For the purposes of processing deposits and withdrawals, only the assets and shares contributed by those requests should be accounted for
- For the purpose of computing and distributing yield only shares that are not part of requests should be accounted

This creates an interesting dynamic where based on the order in which operations are processed, accounting, yield and resulting shares and assets can change

My 2 cents:

- Processing withdrawals before a price update, ensures the shares will not gain yield they may no longer deserve, but creates the risk that these shares will avoid receiving losses

- Processing deposits before a price update will either result in granting these new shares some extra yield or some extra losses, whereas processing them after will grant them either a cheaper share price or a more expensive one

- Processing yield and valuation updates while outstanding shares are part of the accounting will result in either socializing losses or socializing gains to them

I generally believe that given that the admin has the ability to know which value the valuation will take, they should be able to decide which process to follow

I'm not fully confident whether we can argue that there is an ideal process, but rather that formalizing the process that the admin will take will ensure consistent results and prevent abuse

A-02 2 aspects to Pricing Shares when dealing with multi currencies

When dealing with multi-currencies prices of a pool are tied to 2 variables instead of 1:

- The rate of the currencies, between the DepositAsset and the PoolAsset
- The Price Per Share due to appreciation or depreciation of the Pool Currency

It's important you generally avoid comingling these two ratios as the Price Per Share is fairly reliable

Whereas the ratio between currencies is subject to fluctuation and can lead to an issue called: Single Sided Exposure, which happens whenever you try to denominate a pool currency in some other currency and instead of passing that risk and losses to the user, you socialize it to the vault

A-03 Collection of CEI non conforming code when including malicious hooks

Instances

Async Requests

https://github.com/centrifuge/protocol-v3/blob/7fbcf677dab86d94e07c1feea62d387750baa019/src/vaults/AsyncRequests.sol#L368-L395

```
    function mint(address vaultAddr, uint256 shares, address receiver, address controller)
        public
        auth
        returns (uint256 assets)
    {
        AsyncInvestmentState storage state = investments[vaultAddr][controller];
        uint128 shares_ = shares.toUint128();
        _processDeposit(state, shares_, shares_, vaultAddr, receiver); /// @audit Hook into User

        assets = uint256(_calculateAssets(vaultAddr, shares_, state.depositPrice,
MathLib.Rounding.Down)); /// @audit manipulate price in some way?
    }

    function _processDeposit(
        AsyncInvestmentState storage state,
        uint128 sharesUp,
        uint128 sharesDown,
        address vaultAddr,
        address receiver
    ) internal {
        require(sharesUp <= state.maxMint, ExceedsDepositLimits());
        state.maxMint = state.maxMint > sharesUp ? state.maxMint - sharesUp : 0;
        if (sharesDown > 0) {
            require(
                IERC20(IAsyncVault(vaultAddr).share()).transferFrom(address(escrow), receiver,
sharesDown),
                ShareTokenTransferFailed()
            );
        }
    }
```

Balance Sheet

https://github.com/centrifuge/protocol-v3/blob/7fbcf677dab86d94e07c1feea62d387750baa019/src/vaults/BalanceSheet.sol#L203-L224

```
    function _withdraw(
        PoolId poolId,
        ShareClassId scId,
        AssetId assetId,
        address asset,
        uint256 tokenId,
        address receiver,
        uint128 amount,
        D18 pricePoolPerAsset
    ) internal {
        escrow.withdraw(asset, tokenId, poolId.raw(), scId.raw(), amount);

        if (tokenId == 0) {
            SafeTransferLib.safeTransferFrom(asset, address(escrow), receiver, amount);
        } else {
            IERC6909(asset).transferFrom(address(escrow), receiver, tokenId, amount);
        }

        emit Withdraw(poolId, scId, asset, tokenId, receiver, amount, pricePoolPerAsset,
uint64(block.timestamp));

        sender.sendUpdateHoldingAmount(poolId, scId, assetId, receiver, amount, pricePoolPerAsset,
false);
    }
```

https://github.com/centrifuge/protocol-v3/blob/7fbcf677dab86d94e07c1feea62d387750baa019/src/vaults/BalanceSheet.sol#L226-L248

```
    function _deposit(
        PoolId poolId,
        ShareClassId scId,
        AssetId assetId,
        address asset,
        uint256 tokenId,
        address provider,
        uint128 amount,
        D18 pricePoolPerAsset
    ) internal {
        escrow.pendingDepositIncrease(asset, tokenId, poolId.raw(), scId.raw(), amount);

        if (tokenId == 0) {
            SafeTransferLib.safeTransferFrom(asset, provider, address(escrow), amount);
        } else {
            IERC6909(asset).transferFrom(provider, address(escrow), tokenId, amount);
        }

        emit Deposit(poolId, scId, asset, tokenId, provider, amount, pricePoolPerAsset,
uint64(block.timestamp));

        escrow.deposit(asset, tokenId, poolId.raw(), scId.raw(), amount);
        sender.sendUpdateHoldingAmount(poolId, scId, assetId, provider, amount, pricePoolPerAsset,
true);
    }
```

Pool Manager

https://github.com/centrifuge/protocol-v3/blob/7fbcf677dab86d94e07c1feea62d387750baa019/src/vaults/PoolManager.sol#L105-L129

```
    function transferShares(uint16 centrifugeId, uint64 poolId, bytes16 scId, bytes32 receiver, uint128 amount)
        external
        payable
    {
        IShareToken share = IShareToken(shareToken(poolId, scId));
        require(
            share.checkTransferRestriction(msg.sender, address(uint160(centrifugeId)), amount),
            CrossChainTransferNotAllowed()
        );

        gateway.payTransaction{value: msg.value}(msg.sender);

        try share.authTransferFrom(msg.sender, msg.sender, address(this), amount) returns (bool) {}
        catch {
            // Support share class tokens that block authTransferFrom. In this case ERC20 approval needs to be set
            require(share.transferFrom(msg.sender, address(this), amount), TransferFromFailed());
        }

        share.burn(address(this), amount);

        emit TransferShares(centrifugeId, poolId, scId, msg.sender, receiver, amount);

        sender.sendTransferShares(centrifugeId, PoolId.wrap(poolId), ShareClassId.wrap(scId), receiver, amount);
    }
```

Comment

In general you should safely be able to change the code to reorder the sequences to:

- Checks
- Effects
- Internal Interactions (Contracts that are known)
- External Interactions (Contracts that are not known)

In the case of `AsyncRequests.mint` this could lead to using different prices, (although arguably reentrancy wouldn't be necessary for this)

The refactoring would have you move all transfers at the end of functions

Interacting with Balance Sheet and Sender at the end of the function

It's worth noting that the try catch doesn't check for gas, meaning that putting it at the end of the function makes it more likely to be subject to #7

A-04 `ShareClassManager.claimDepositUntilEpoch` can open up Race Condition in `claimXUntilEpoch` can create ghost `pendingX` due to calling `_postClaimUpdateQueued` when epochs are still being proceesed

Impact

The following impact should only be possible when `claimDepositUntilEpoch` is callable by a user When such a scenario arises, this becomes possible:

Post loop auto-queueing

Instead of being forced to claim all, you claim only a portion

Either if calling `claimDepositUntilEpoch` is possible

Or if you call `claimDeposit` when messages about future epochs have already been broadcasted and reverted or are pending

You can then: `claimDepositUntilEpoch`

Which will process queued into the global `pendingDeposit`

This will update the `userOrder.pending` (which is not tied to any epoch)

This will keep the global `pendingDeposit` to have the `userOrder.pending`

But the user will be able to claim during the current epoch at the current `epochAmounts[scId_][depositAssetId][epochId_]`

This will break the invariant of the `pendingDeposit` being the sum of `userOrder.pending`

TODO: POC

Request 1: Epoch 1 - 100 - Approved later [User A]

Request 2: Epoch 1 (is queued) - 100 [User A]

Request 3: Epoch 2 - 100 - Approved [User B]

User A Can chose to process their approval of Request 1, this will cause the queue to also be added to pending, and result in a non-zero `userOrder.pending`

Since Request 3 was processed with user B amounts

But User A has pending amounts, they can claim their remaining `pending` via the epoch amounts added when processing request 3

This breaks a key invariant that the:

- Sum of all processable amounts in an epoch is at most equal to the amount of `pendingX`

Breaking the invariant would leave `pendingX` to have the additional amount from the queued amounts caused by User A

This would leave these "ghost pending" amounts permanently in the SCM

Mitigation

I believe that `claimDepositUntilEpoch` should be made internal and should never be callable

This race condition should be documented and prevented

A-05 Suggested Next Steps

Executive Summary

Over the review I have looked into

- Flow of Accounting and Holdings for both systems (Mostly covered)
- Analysis of State Transitions for Holdings and Accounting (Covered Extensively)
- Analysis of SCM Edge Cases (Covered Extensively)
- Analysis on Price and Rounding Error (Covered Extensive)
- Analysis on Assets and Solvency for Vaults (Covered)
- Analysis of Rounding Directions (Covered Extensively)
- Flow of Messages (Not covered)

NOTES: I was originally scheduled to perform a 2 weeks review on the codebase, after a few days I've asked to extend the timeline as I believe that some parts of the code are pretty complex to understand and to explore with automated tooling

CONTEXT: During the review, over 30 different commits and changes were pushed to the Repo, changes cause a degradation to the relevance of many findings

Overall I believe that for the next review, a zero change policy should be put into place

In order to achieve it you should:

- Review all findings I sent
- Ensure all parts of the codebase are tested, with libraries being FVd and intra-component state-transitions being proven inductively
- Do another pass on the code to simplify it, document the key features of it (Admin Operations, User Operations, Flow of funds, Flow of Value)

META NOTES: The code is missing one pass to simplify it and make it easier to understand I believe this type of pass won't necessarily improve the code security, but it will improve the ability for new people to look at the code and it will also help auditors be more efficient

Vaults Pricing Flow of funds SCM rounding error (so we are very confident)

Things I overlooked

I worked under the assumption that all encoding and decoding of messages is done with Formally Verified Libraries

I did not check the Holdings flow when the account is a Liability

I did not check every instance of EIP conformance

Documentation

Document every intended flows with the goal of helping auditors identify unintended flows

Document invariants or properties based on specific use cases, the code allows for many possible combinations, it's getting hard to discern between an admin mistake and an intended "irregular transition", better documenting the use cases can help flag unintended behaviour

Document all messaging flows, as a means to highlight the linear connection between the various parts of the code

Document the admin privileges you plan on setting up

Before the next audit

Make sure 100% of the code is done, and fully tested as well as documented

I believe that given some of the complexity in the system, you will benefit by having as much information as possible given to reviewers as they will need spend quite some time to fully understand how the system works and how the various components are related

Additionally, make sure to run Invariant Suites extensively, leverage Corpus Reuse to make the runs faster, and send every edge case to auditors for them to review

Investigating Precision Loss and Rounding Errors

Consider using this repo: https://github.com/Recon-Fuzz/centrifuge-hack

Setting up custom optimization tests is pretty easy especially if you autocomplete with AI

Ultimately an optimization test just needs to return a int256

Thanks to these, you can explore edge cases that are more realistic

Unfortunately this process takes quite some time, however it's going to help you decide if you should change the pricing logic or accept it's limitation as known

Key considerations for future review

There are many behaviours that can be performed that would be considered "unintended" but there are just as many behaviour that are non-obvious that would be considered "intended"

A lack of a clear FSM opens up to an almost infinite amount of combinations that are very hard to track

Invariant testing will be limited to specific behaviours that are sound

The limitation will open up to possible edge cases in production, monitoring these edge cases will be as important as actually preventing them

Invariant Testing Suites you can reuse

Simplified Testers

- SCM tester: https://github.com/Recon-Fuzz/centrifuge-scm-simplified-fuzz/blob/main/test/recon/Properties.sol
- Pricing Tester: https://github.com/Recon-Fuzz/centrifuge-hack
- Hub Accounting and Holdings Tester: https://github.com/Recon-Fuzz/centrifuge-hub-simplified-fuzz/blob/main/test/pools/fuzzing/recon-scm-only/TargetFunctions.sol

Full Testers (Nicanor)

- Hub Tester: https://github.com/centrifuge/protocol-v3/tree/feat/recon-invariants/test/hub/fuzzing
- Centrifuge Vaults: https://github.com/centrifuge/protocol-v3/tree/feat/recon-invariants/test/vaults/fuzzing

A-06 An auditor introduction to the Hub Codebase and its relation to Vaults

Executive Summary

These are notes and comments to help onboard new auditors into the codebase

As of today (10th April) the code is hardened in many areas, however it lacks on layer of polish to make it a bit simpler

My goal with this document is to help distil some key insights that would help simplify the code

And to onboard new reviewers to ensure the code is safe from exploits

Hub Structure

├── Hub.sol (Main hub contract) ├── HubRegistry.sol (Registry management) ├── ShareClassManager.sol (Share class management) ├── Holdings.sol (Asset holdings management) ├── Accounting.sol (Accounting operations)

The Hub is the core pipe contract, that ensures all other contracts are in synch

IMO you should check this multiple times, with the expectation that you will fully understand how it works only on the last pass

Hub Part 1

Create Pool

As a trusted actor you can call `createPool` to create a new pool

Create Accounting

Call `createAccount` for the 4 accounts you will use

This will keep track of the 4 accounts separately 3 accounts can be used to derive the 4th, see below The relation between these accounts can be used to derive the Value Held

Create Holdings

Create the Holdings after having created the 4 accounts

This will allow to record effectively a delta between the value of what was added and what was removed

Accounting

Is a contract used to track debit and credits pertaining to a specific account

As of now there are 6 accounts:

- Asset

- Equity
- Loss
- Gain
- Expense
- Liability

It's worth noting that doing nothing is not the same as doing something, as adding and removing is done via `addCredit` / `addDebit`

Accounting Flows

- `approveDeposits` | +C Equity | +D Asset
- `revokeShares` | +C Asset | +D Equity

On a Positive `holdings.update` If holdings is not a liability

- +C Gain | +D Asset

On a Negative `holdings.update` If holdings is not a liability

- +C Asset | +D Loss

Reasoning for non liability holdings

For holdings that are not a liability we have 4 accounts:

- Asset
- Equity
- Loss
- Gain

Out of all these 4 Asset is the only one that is DebtNormal

Meaning an increase in Debt, is an increase in value

BASE DEPOSIT

When add currency, I increase the Credit of my Equity and increase the Debit of Assets When I remove it, the opposite happens, I increase the Credit of Assets and increase Debit of Equity

YIELD CHANGE

When yield is obtained, our Equity is unchanged (what put in) Our Assets do change (the value of what we hold) We Add Credit to Gain (It has increased as we expected) And Add Debit to Asset (cause it's debt normal, so adding Debit is adding value)

When value is lost Our Equity is unchanged We add Credit to Asset (cause it's debt normal, the value is being lost) And we add Debit to Loss (cause Loss is tracking real losses, it's Credit Normal so this is an actual negative value)

SO FAR

NOTE: Technically it's always Debit - Credit or Credit - Debit, see `accountValue`

Debit of Asset = Value you have Credit of Equity = Value you put in Credit of Gain = Yield you earned Debit of Loss = Value Losses

---

PARTIAL UPDATE

A partial update is effectively a balanced change of values, triggered by `updateHoldingAmount`

Accounting Properties

Accounting must be balanced

The presence of `addDebit` and `addCredit` threaten this property as well as the ability to dump values into a non existent account

However, if we assume these functions are never used maliciously, the rest of the function seems balanced, and follows the rules above

As discussed it's worth noting that a 0 can either be a 0 or a Debit - Credit == 0 (or vice versa for Credit Normal Accounts)

Holdings

This is a pretty straightforward contract which tracks the values that a PID + ScID have received

It's worth noting that the values are the "sum of segments" of the amount and value at a time, the values are reconciled (triggering changes in Accounting) by calling `update`

This leads to absolute values always matching, but relative values could be different

That's worth monitoring over time as the "unupdated" sum of segments will roughly match the relation of EQUITY vs ASSETS

Where the sum of segments should math EQUITY if no `holdings.update` is ever called

And it should match ASSETS whenever `holdings.update` is called (triggering a GAIN or a LOSS)

HubRegistry

Effectively just used to ensure pools exists as well as manager, and currency for pools

ShareClassManager

Our starting point as a user is in requestingDeposit / Redeem and optionally cancelling it

See: https://github.com/Recon-Fuzz/centrifuge-review-protocol-v3/issues/21 https://github.com/Recon-Fuzz/centrifuge-review-protocol-v3/issues/19

Combining these can result in interesting states, which you should explore

From my POV once a deposit has been requested we can basically only enter a cancellation request Or we can have those new requests be queued

Queued requests are effectively "separated" from all logic, the queueing ensures that an approval on `approveDeposits`

Is done on the expected amount of assets

It is worth noting that Queueing doesn't provide any particular security guarantee as you can sidestep it by using other accounts, also previously queued deposits don't force a follow up deposit to be queued, meaning they effectly allow the sidestepping one epoch after

`approveDeposits` is confirming the amount of assets that the manager will use to purchase underlying shares

Whereas `issuingShares` will have a confirmed `navPerShare` for each share, meaning that it's locking in the ratio between the assets added and the shares outstanding

Note that this can have fluctuations in the nav, since this is changing the `navPerShare`

This introduces rounding errors both in the valuation of shares in the vaults as well as the valuation of assets / shares, although the admin should have all tools to balance out these imprecisions

Expected formula for evaluating holdings

Holdings Total Value = Sum of Segments Holdings Value after an update = Amt * Valuation

Delta change after an update Positive: Credit in Gain, Debit in Asset

Gain Credit - Debit = Debit growth of Asset over Equity

Negative: Credit Asset, Debit in Loss Loss Debit - Credit = Credit growth of Asset over Equity (= Loss)

Total Value (Value of Holdings) accountValue(Asset)

Total Deposits (Total amount deposited) -> Value / Deposit = Implied PPFS Appreciation (TODO) accountValue(Equity)

Total Yield (Includes losses): (NOTE: Can overflow) accountValue(Asset) - accountValue(Equity)

ASSETS: accountValue(Equity) + accountValue(Gain) + accountValue(Loss) // Assuming loss is a negative value

EQUITY: accountValue(Asset) + accountValue(Loss) - accountValue(Gain) // Loss comes back, gain is subtracted

GAIN Total Yield + accountValue(loss) /// I had to gain yield + loss to get to yield

LOSS Total Yield (abs) - accountValue(gain) // Negative Loss (- of this is loss)

Vaults and Hub

The relation between vaults and hub is the following:

- Vaults will call Escrow and Balance Sheet

Balance Sheet is connecting Hub by sending and receiving messages that synch the Assets and Liabilities

Escrow is holding the assets Vaults have Shares, Amounts and Prices

Through Vaults shares, amounts and prices, the vault can determine how to distribute the assets and shares from the Escrow Some of these operations call the Balance Sheet as they have an impact on the total amount of assets (TODO CHECK)

Asynchronous Vaults Flow

The Asynchronous operations work in a way that matches the V2 implementation:

- Request OP (deposit / redeem)
- Approve OP (deposit / redeem)
- issue / revoke shares (locks in the values in Hub and unlocks them in the Escrow)
- OP (deposit|mint / redeem | withdraw)

These tend to not only follow the same steps but ultimately follow a back and forth mechanism that is consistent with V2

The security properties around these operations are the following:

- A user receives a custom price, the sum of all prices tends to match the total value in the pool, but due to precision, extra fees, and race conditions, the sum can be slightly different, it's important that this discrepancy is monitored and that you make sure it's very limited in it's impact

Synchronous Vault Flows

For security reasons (ability to liquidate profits), redemptions and withdrawals do not have a Synchronous Option

I generally agree with the decision because of the fact that exiting a Vault can lock in losses to it

I've discussed the risks of Synchronous Operations in #35

Synchronous Deposits are effectively a Macro, with simplification of the:

- Request
- Approve
- Issue
- OP

Flow as they ultimately perform all of them

The reason why this can work is that the price would be cached, this can cause issues as it possibly leaks value through #35 But most importantly this creates a scenario where the admin is liable to have to actually purchase the underlying asset, after it has been sold

This will most likely require using a cached price that is higher than what would be the current spot price as to ensure that the manager is not liable for more value than intended

So ultimately sync operations are moving some of the risk to the manager, as long as repricing of the underlying asset is done in a way that is consistent and doesn't have step-wise changes, then the risk to the manager should be predictable and possibly avoidable

It's also worth noting that adding minting fees (which could already be added by issuing more shares, or reducing the assets), would possibly compensate if not outright eliminate this risk

Balance Sheet and Vault

Synchronous Deposits are macros for: Request Deposit -> Approve Deposit -> Issue Shares -> Deposit, as such they use `BalanceSheet.noteDeposit` as well as `BalanceSheet.issue`

Asynchronous Deposits instead simply trigger `sender.sendDepositRequest` without causing changes to the balance sheet This is because the change will happen once the deposit is approved via `BalanceSheet.approvedDeposits`

Withdrawals all go through RequestRedeem which follows a similar mechanism

However, because of #44 withdraw is changing the accounting instead of `revokeShares` which I believe is not correct

Escrow and Vault

The escrow handles all of the assets, whether these are being deposited, withdrawn or cancelled

Assets part of Requests are sent to the escrow but they are not added to storage (asynchronous requests) Whereas assets that are part of Synchronous deposits are added to the Escrow and Balance Sheet (but the issued shares will temporarily be unbacked) Assets that are being "protected" as part of withdrawals go to the `reserve` via `escrow.reserveIncrease` Asset that are deposited go to the escrow and when accounted will be added to the Balance Sheet Assets that are to be withdrawn by users are Reserved and will then go through a Withdrawal Assets that are withdrawn by the admin go through withdraw without being reserved

It's critical to note how brittle the connection between the Escrow, Balance Sheet and Holding is unless the Manager does a concerted effort to maintain it

In lack of that effort it will be very difficult for any reviewer to quickly get up to speed

E-01 Sync Deposit / Mint can open up to risk free arbitrage when a positive price update is known

Impact

`SyncRequests` allow `mint`ing and `deposit`ing without any delay

The price of the shares is based on a cached value, the update of which could be front-run, especially when said update would be triggered from another chain

Mitigation

Because we rely on external pricing there is no clear mitigation to this issue

Other vaults have added:

- Fees as a means to make arbitrage unprofitable
- Yield Profit Decay as a means to ensure that a depositor would socialize sufficient yield to the vault before taking some

Ensuring the vault can only mint up to a cap can also reduce risk

I-01 New Property Checks for Stateful Fuzzing

Accounting

- `accountValue` should never revert
- Convert the `accountValue` formula to use `uint128` and compare against `int128`, the results should always match (note that you need to account for revert cases which requires some nuance)
- account.totalDebit and account.totalCredit is always less than `uint128(type(int128).max)` (this should be 1 less than the absolute value of `int128.min` which can open up another edge case
- Accounting should always be balanced, the only way to imbalance it is for the admin to dump values into a fictitious unrecognized account. They may do so to fix errors, or as a mistake

Holdings

`update` should never revert with `Int128_Overflow`

Hub

- Any decrease in valuation (holdings.update) should not result in an increase in `accountValue`
- Total Sum of Segments (tedious to code, but we already did it for eBTC)

Optimization Tests

- Optimize the maximum loss for user claimable vs requested
- Optimize the maximum dust amount of requested deposits and redemptions that can never be claimed

Doomdsay

If everyone claims we have X amt of unclaimable If we claim again in the future the unclaimable amount must be GTE X

Hub -> Accounting Soundness Properties

```
// Holdings property should be the following:
    // Sum of Deposit - Sum of withdrawals =
    // Sum of ASSETS
    function property_trackingOfAmounts() public {
        (uint128 amt, ) = _getAmountAndValue();
        eq(amt, depositAmt, "property_trackingOfAmounts");
    }
    function property_trackingOfValues() public {
        (, uint128 value) = _getAmountAndValue();
        eq(value, depositValue, "property_trackingOfValues");
    }

    function _getAmountAndValue() internal view returns (uint128, uint128) {
        (uint128 assetAmount, uint128 assetAmountValue, ,) = holdings.holding(poolId, scId,
payoutAssetId);
        return (assetAmount, assetAmountValue);
    }

    // SUM OF VALUE

    /// FOUNDATIONAL PROPERTIES
    // Soundness
    function property_sound_loss() public {
        t(accounting.accountValue(poolId, LOSS_ACCOUNT) <= 0, "Loss is always negative");
    }
    function property_sound_gain() public {
        t(accounting.accountValue(poolId, GAIN_ACCOUNT) >= 0, "Gain is always positive");
    }

    /// SUM of accounting
    function property_sum_of_losses() public {
        t(lossValue == int256(accounting.accountValue(poolId, LOSS_ACCOUNT)),
"property_sum_of_losses");
    }

    function property_sum_of_gains() public {
        t(yieldValue == int256(accounting.accountValue(poolId, GAIN_ACCOUNT)),
"property_sum_of_gains");
    }
```

Hub -> Accounting type properties

```
   // Function global solvency property
    // Equity | Assets = Either Yield or Loss
    // Equity = Assets + Yield - Loss
    // But not just as accounts, more

    // Accounting Properties | TODO
    function property_total_yield() public {
        int128 assets = accounting.accountValue(poolId, ASSET_ACCOUNT);
        int128 equity = accounting.accountValue(poolId, EQUITY_ACCOUNT);

        if(assets > equity) {
            // Yield
            int128 yield = accounting.accountValue(poolId, GAIN_ACCOUNT);
            t(yield == assets - equity, "property_total_yield gain");
        } else if (assets < equity) {
            // Loss
            int128 loss = accounting.accountValue(poolId, LOSS_ACCOUNT);
            t(loss == assets - equity, "property_total_yield loss"); // Loss is negative
        }
    }

    function property_asset_soundness() public {
        int128 assets = accounting.accountValue(poolId, ASSET_ACCOUNT);
        int128 equity = accounting.accountValue(poolId, EQUITY_ACCOUNT);
        int128 loss = accounting.accountValue(poolId, LOSS_ACCOUNT);
        int128 gain = accounting.accountValue(poolId, GAIN_ACCOUNT);

        // assets = accountValue(Equity) + accountValue(Gain) - accountValue(Loss)
        t(assets == equity + gain + loss, "property_asset_soundness"); // Loss is already negative
    }

    function property_equity_soundness() public {
        int128 assets = accounting.accountValue(poolId, ASSET_ACCOUNT);
        int128 equity = accounting.accountValue(poolId, EQUITY_ACCOUNT);
        int128 loss = accounting.accountValue(poolId, LOSS_ACCOUNT);
        int128 gain = accounting.accountValue(poolId, GAIN_ACCOUNT);

        // equity = accountValue(Asset) + (ABS(accountValue(Loss)) - accountValue(Gain) // Loss comes back, gain is subtracted
        t(equity == assets + (-loss) - gain, "property_equity_soundness"); // Loss comes back, gain is subtracted, since loss is negative we need to negate it
    }

    function property_gain_soundness() public {
        int128 assets = accounting.accountValue(poolId, ASSET_ACCOUNT);
        int128 equity = accounting.accountValue(poolId, EQUITY_ACCOUNT);
        int128 loss = accounting.accountValue(poolId, LOSS_ACCOUNT);
        int128 gain = accounting.accountValue(poolId, GAIN_ACCOUNT);

        // Total Yield = // accountValue(Asset) - accountValue(Equity))
        // Gain = Total Yield + accountValue(loss) /// I had to gain yield + loss to get to yield
        int128 totalYield = assets - equity; // Can be positive or negative
        t(gain == totalYield + (-loss), "property_gain_soundness");
    }

    function property_loss_soundness() public {
        int128 assets = accounting.accountValue(poolId, ASSET_ACCOUNT);
        int128 equity = accounting.accountValue(poolId, EQUITY_ACCOUNT);
        int128 loss = accounting.accountValue(poolId, LOSS_ACCOUNT);
        int128 gain = accounting.accountValue(poolId, GAIN_ACCOUNT);
        // Loss = Total Yield (abs) - accountValue(gain) // Negative Loss (- of this is loss)
        int128 totalYield = assets - equity; // Can be positive or negative
        t(loss == totalYield - gain, "property_gain_soundness");
    }

    function property_accounting_and_holdings_soundness() public {
        // Accounting.assets is the value held
        // Holdings.value is the value held, they must match
        uint128 assets = uint128(accounting.accountValue(poolId, ASSET_ACCOUNT));
        uint128 holdingsValue = holdings.value(poolId, scId, depositAssetId);

        // This property holds all of the system accounting together
        eq(assets, holdingsValue, "Assets and Holdingsm must match");
    }
```

Vaults

SOLVENCY - This is always lte the actual assets

https://github.com/centrifuge/protocol-v3/blob/3b7a4bc227dab89561efc5f6e4983ce691c08284/src/vaults/BaseVaults.sol#L163-L165

```
    function totalAssets() external view returns (uint256) { /// @audit Rounds down the assets, so this can prob only be proven with a bound
        return convertToAssets(IERC20Metadata(share).totalSupply());
    }
```

And optimize to prove the loss is only up to 1 share - 1

SOLVENCY PROPERTY CAN ONLY INCREASE - Any operation that causes losses, causes losses to the user not to the system

TODO: Limit losses to users to an order of magnitude that is manageable

DOOMSDAY - Price Per Share Property Test

https://github.com/centrifuge/protocol-v3/blob/3b7a4bc227dab89561efc5f6e4983ce691c08284/src/vaults/BaseVaults.sol#L183-L185

```
    function pricePerShare() external view returns (uint256) {
        return convertToAssets(10 ** _shareDecimals);
    }
```

I pay Price Per Share + PRECISION

I receive Price Per Share - PRECISION

DOOMSDAY - After USER OP - PPFS NEVER CHANGES!

pricePerShare -> Static

DOOMSDAY - After Us OP - Implied PPFS never changes (the ratio of total assets and shares, as opposed to the ppfs which is hardcoded / from CP)

DOOMSDAY - Won't revert due to roundUp - No user OP should ever revert with any value within the bounds (assuming not paused, not blacklisted, etc..)

https://github.com/centrifuge/protocol-v3/blob/3b7a4bc227dab89561efc5f6e4983ce691c08284/src/vaults/AsyncRequests.sol#L377-L392

```
    function _processDeposit(
        AsyncInvestmentState storage state,
        uint128 sharesUp,
        uint128 sharesDown,
        address vaultAddr,
        address receiver
    ) internal {
        require(sharesUp <= state.maxMint, ExceedsDepositLimits());
        state.maxMint = state.maxMint > sharesUp ? state.maxMint - sharesUp : 0;
        if (sharesDown > 0) {
            require(
                IERC20(IAsyncVault(vaultAddr).share()).transferFrom(address(escrow), receiver,
sharesDown),
                ShareTokenTransferFailed()
            );
        }
    }
```

Price Per Share Overall Invariant

The price per share used in the entire system is ALWAYS provided by the admin

(Could store the last value used and then check against other calls)

ERC7540 New Properties

- I can always maxDeposit, maxMint, maxRedeem, maxWithdraw if it's non-zero and I'm approved
- I can do the above with a value that is between 1 and the max
- I can do the above and the remainder is exactly the amt left (Prob can catch some rounding error here)

Depositing maxDeposit leaves me with 0 orders Depositing Max never surpasses maxMint Minting max mint leaves me with zero maxMint Minting Max never surpasses maxDeposit (NOTE: Same for Redeem and Withdraw)

Cross Chain Solvency

The sum of assets (assetId) on all Escrows on all chains (availableBalanceOf), for each PoolId, ScID Is equal to the total in the Holdings for each PoolId, ScID

Fundamentally:

- I can requestDeposit on some other chain
- This can be approved
- The admin can take the assets, bridge them, burn them, etc...
- The assets should eventually go into the Holdings as another assetId (or the same)
- The assets should eventually go to one of the escrows, likely the "canonical" one

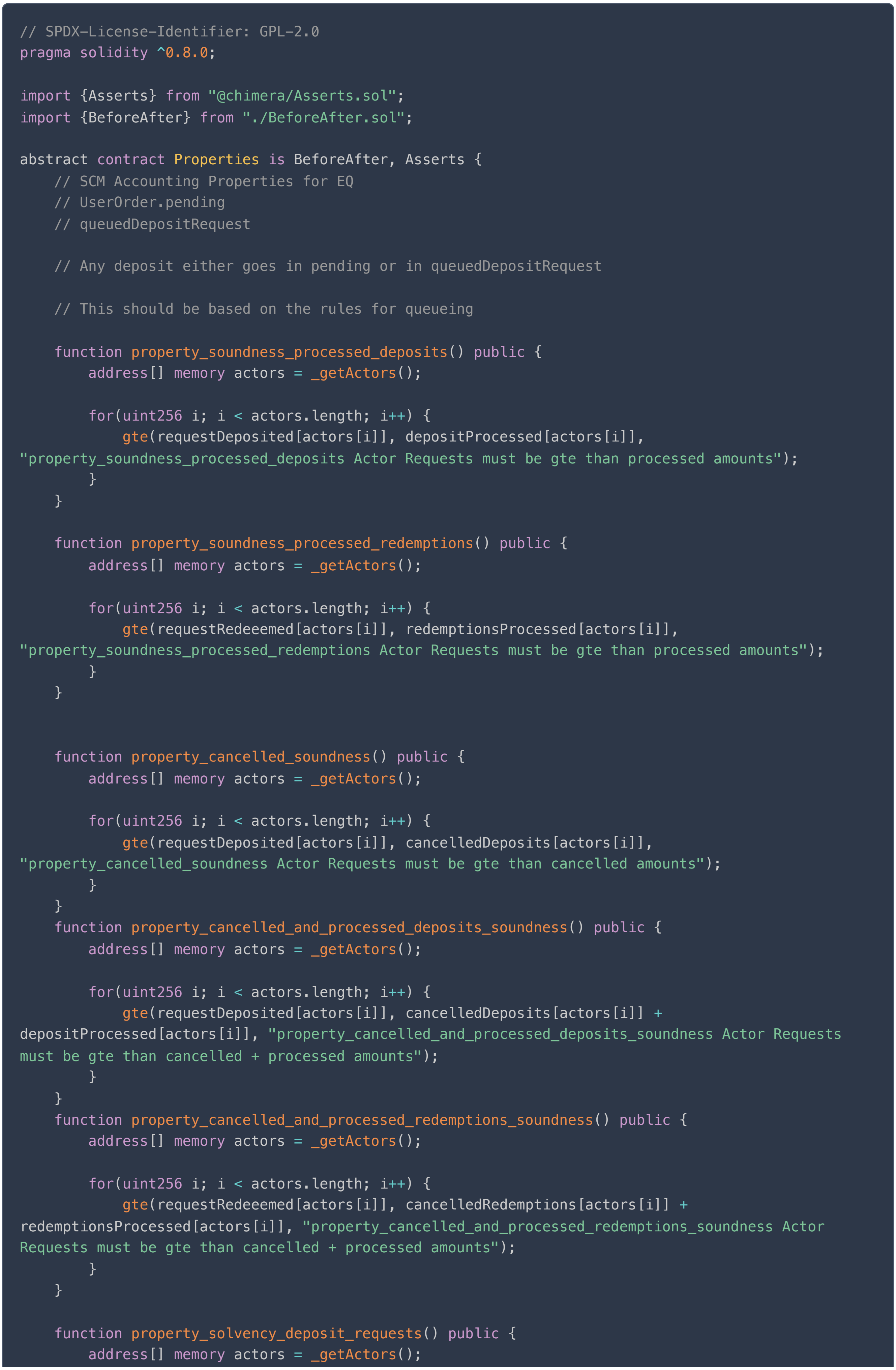
This property breaks if:

- The admin doesn't follow the state transitions we wrote for the simplified hub tester

Meaning this can be tested only if we hardcode most state transitions and behaviour

Scm Properties

See: https://github.com/Recon-Fuzz/centrifuge-scm-simplified-fuzz/blob/main/test/recon/Properties.sol



```
// SPDX-License-Identifier: GPL-2.0
pragma solidity ^0.8.0;

import {Asserts} from "@chimera/Asserts.sol";
import {BeforeAfter} from "./BeforeAfter.sol";

abstract contract Properties is BeforeAfter, Asserts {
    // SCM Accounting Properties for EQ
    // UserOrder.pending
    // queuedDepositRequest

    // Any deposit either goes in pending or in queuedDepositRequest

    // This should be based on the rules for queueing

    function property_soundness_processed_deposits() public {
        address[] memory actors = _getActors();

        for(uint256 i; i < actors.length; i++) {
            gte(requestDeposited[actors[i]], depositProcessed[actors[i]],
"property_soundness_processed_deposits Actor Requests must be gte than processed amounts");
        }
    }

    function property_soundness_processed_redemptions() public {
        address[] memory actors = _getActors();

        for(uint256 i; i < actors.length; i++) {
            gte(requestRedeeemed[actors[i]], redemptionsProcessed[actors[i]],
"property_soundness_processed_redemptions Actor Requests must be gte than processed amounts");
        }
    }


    function property_cancelled_soundness() public {
        address[] memory actors = _getActors();

        for(uint256 i; i < actors.length; i++) {
            gte(requestDeposited[actors[i]], cancelledDeposits[actors[i]],
"property_cancelled_soundness Actor Requests must be gte than cancelled amounts");
        }
    }
    function property_cancelled_and_processed_deposits_soundness() public {
        address[] memory actors = _getActors();

        for(uint256 i; i < actors.length; i++) {
            gte(requestDeposited[actors[i]], cancelledDeposits[actors[i]] +
depositProcessed[actors[i]], "property_cancelled_and_processed_deposits_soundness Actor Requests
must be gte than cancelled + processed amounts");
        }
    }
    function property_cancelled_and_processed_redemptions_soundness() public {
        address[] memory actors = _getActors();

        for(uint256 i; i < actors.length; i++) {
            gte(requestRedeeemed[actors[i]], cancelledRedemptions[actors[i]] +
redemptionsProcessed[actors[i]], "property_cancelled_and_processed_redemptions_soundness Actor
Requests must be gte than cancelled + processed amounts");
        }
    }

    function property_solvency_deposit_requests() public {
        address[] memory actors = _getActors();
        uint256 totalDeposits;

        for(uint256 i; i < actors.length; i++) {
            totalDeposits += requestDeposited[actors[i]];
        }

        gte(totalDeposits, approvedDeposits, "Total Deposits must always be less than
totalDeposits");
    }

    function property_solvency_redemption_requests() public {
        address[] memory actors = _getActors();
        uint256 totalRedemptions;

        for(uint256 i; i < actors.length; i++) {
            totalRedemptions += requestRedeeemed[actors[i]];
        }

        gte(totalRedemptions, approvedRedemptions, "Total Redemptions must always be less than
approvedRedemptions");
    }


    // At any time you can  never request to redeem more than total supply
    // You also can never ask to redeem more than what you have


    function property_actor_pending_and_queued_deposits() public {
        // Pending + Queued = Deposited?
        address[] memory actors = _getActors();

        for(uint256 i; i < actors.length; i++) {
            (uint128 pending, ) = shareClassManager.depositRequest(scId_, paymentAssetId,
_asBytes(actors[i]));
            (, uint128 queued) = shareClassManager.queuedDepositRequest(scId_, paymentAssetId,
_asBytes(actors[i]));

            // user order pending
            // user order amount

            // NOTE: We are missign the cancellation part, we're assuming that won't matter but idk
            eq(requestDeposited[actors[i]] - cancelledDeposits[actors[i]] -
depositProcessed[actors[i]], pending + queued, "property_actor_pending_and_queued_deposits");
        }
    }
    function property_actor_pending_and_queued_redemptions() public {
        // Pending + Queued = Deposited?
        address[] memory actors = _getActors();

        for(uint256 i; i < actors.length; i++) {
            (uint128 pending, ) = shareClassManager.redeemRequest(scId_, paymentAssetId,
_asBytes(actors[i]));
            (, uint128 queued) = shareClassManager.queuedRedeemRequest(scId_, paymentAssetId,
_asBytes(actors[i]));

            // user order pending
            // user order amount

            // NOTE: We are missign the cancellation part, we're assuming that won't matter but idk
            eq(requestRedeeemed[actors[i]] - cancelledRedemptions[actors[i]] -
redemptionsProcessed[actors[i]], pending + queued, "property_actor_pending_and_queued_redemptions");
        }
    }
```

```
}
```

Price Full Math Precision Loss Optimization Study

https://github.com/Recon-Fuzz/centrifuge-hack

Tests I didn't write

- Doomsday for SCM (overflow reverts, DOS, permanent DOS)
- Insolvency in Escrow vs Holdings

Additional Services by Recon

Recon offers:

- Invariant Testing Audits - We'll write your invariant tests then perform and audit on the code
- Cloud Fuzzing as a Service - The easiest way to run invariant tests in the cloud - Ask about Recon Pro
- Audits - High Quality Audits done by Highly Qualified Reviewers that work with Alex personally